

Linda-based applicative and imperative process algebras

Rocco De Nicola*, Rosario Pugliese

Dipartimento di Sistemi ed Informatica, Università di Firenze Via Lombroso 6/17, 50134 Firenze, Italy

Received March 1997; revised February 1999

Communicated by J.W. de Bakker

Abstract

The classical algebraic approach to the specification and verification of concurrent systems is tuned to distributed programs that rely on asynchronous communications and permit explicit data exchange. An applicative process algebra, obtained by embedding the Linda primitives for interprocess communication in a CCS/CSP-like language, and an imperative one, obtained from the applicative variant by adding a construct for explicit assignment of values to variables, are introduced. The testing framework is used to define behavioural equivalences for both languages and sound and complete proof systems for them are described together with a fully abstract denotational model (namely, a variant of Strong Acceptance Trees). © 2000 Elsevier Science B.V. All rights reserved.

Keywords: Concurrency; Asynchronous communications; Process algebras; Formal semantics

1. Introduction

The availability of sophisticated parallel hardware at limited costs has led to a proliferation of programming languages aiming at taking advantage of the new computing capabilities. These languages are equipped with primitives for *interprogram communication* and permit designing concurrent and distributed programs. However, this class of programs is difficult to design and debug. The possible interactions between two or more concurrent programs may give rise to new, unwanted, behaviours and may lead to nondeterministic executions.

There have been several efforts to model concurrent programs and to develop methods for reasoning about them. Probably, the most well-known approach is the *process algebraic* one (CCS [37], ACP [7], CSP [32], etc.). The basic idea of process algebras is that distributed systems may be modelled as sets of concurrent communicating

* Corresponding author. Tel.: +39-055-479-6733; fax: +39-055-479-6730.

E-mail addresses: denicola@dsi.unifi.it (R. De Nicola), pugliese@dsi.unifi.it (R. Pugliese)

processes, and the main aim is that of providing both description languages and techniques for assessing correctness. The languages are based on small sets of elementary constructs that permit describing systems at different levels of abstraction. The operators have intuitive interpretations, and model basic notions like parallel composition, nondeterminism, abstraction, sequentialization, etc.

Within the process algebraic approach, both *specifications* (the descriptions of the expected behaviour of systems in terms of their reactions to external stimuli) and *implementations* (the detailed descriptions of systems with information about their logical or physical structures) can be expressed in the same language. There is no absolute distinction between specifications and implementations; within particular settings a program may be considered as a specification while in others it may be used as an implementation description.

The relationships between the different levels are assessed by means of behavioural relations between systems. They can be used to check whether two systems have the “same” behaviour or one is an “approximation of” the other (see, e.g., [5,19,37,39]). Verification consists in studying the relationships between descriptions and implementations. This task may be, at least partially, mechanized either by taking advantage of sets of laws that are consistent with the selected behavioural relation or by “ad hoc” efficient algorithms.

The algebraic approach has so far mainly concentrated on languages with uninterpreted action symbols that rely on a synchronous paradigm for program interaction. Also, the exchange of information between programs has often been limited to synchronization signals and only in few cases languages have been studied that permit explicit data exchange.

In this paper, we take the Linda paradigm for process interaction as our starting point for defining an asynchronous process algebra with explicit data value exchange. Linda [24,12] is a member of a relatively recent generation of global environment parallel languages (e.g. Concurrent Prolog [45], UNITY [14], Shared Prolog [3]) that differ from the previous ones because they offer, and often require, explicit control of interactions. A communication between Linda processes is obtained by accessing tuples (sequences of variables and data) in a shared memory called “tuple space” (a multiset of tuples). The communication mechanism is *asynchronous*, in that send operations are non-blocking, and *associative*, in that tuples are retrieved by referring to (part of) their content; read/receive operations look for tuples with a specific structure and may cause a block.

Our main objective is that of developing a semantic framework that supports analysis of programs written in some *applicative* (without assignment) or *imperative* (with assignment) Linda dialect. The languages we consider in this paper are somehow in between process algebras and the many Linda programming dialects. Linda itself is not a programming language; it is a coordination model whose primitives are devoted to coordinate interactions among programs. Concurrent languages can be obtained by embedding Linda in a sequential (functional, imperative, logic, etc.) programming language (see, e.g., [6,46,40,44]).

We start by introducing a *Process Algebra based on Linda* (PAL), a process algebra obtained by interpreting abstract actions as Linda primitives. Asynchrony is modelled by considering outputs as elementary concurrent processes, whose execution does not delay the progress of the senders. By relying on commutativity and associativity of the operator for parallel composition, the Linda tuple space is rendered by means of parallel composition of (the processes which correspond to) its tuples.

We will use testing preorders [19,28] as the observational machinery for abstracting away from unwanted details of programs and for assessing their correctness by comparing them with respect to a notion of “being an approximation of”. The choice of the observational machinery has been partially dictated by the language; for example, with our operators for modelling non-deterministic situations, bisimulation would fail to be an equivalence relation. The testing approach does not require any adaptation to the language; the observation mechanism directly relies on the communication paradigm.

A behavioural testing (must) preorder is defined, where observers are, like for CCS, processes which can interact with the observed process and report success. Two other equivalent interpretations for PAL processes are given: an equational interpretation via a sound and complete proof system, useful for performing process verification via symbolic manipulation, and a denotational one in terms of double-labelled trees, AT^L (*acceptance trees for Linda*), a generalization of the acceptance trees of [27].

We will also generalize this framework to IPAL, an imperative version of PAL with an assignment command. We shall however maintain that information between processes is exchanged only via the tuple space and shall thus be able to model private stores of processes via explicit substitutions. This choice, and the fact that PAL (and IPAL) binders for value variables and for process variables cannot interfere, enables us to reuse all of the theory developed for PAL to obtain a sound and complete proof system and a fully abstract denotational model for IPAL too. Our semantics for IPAL is a significant simplification of those in [30,23], that require explicitly modelling of processes private stores when adapting the testing scenarios of CCS with value passing (see [29]) and of PAL (see [22]) to extensions with assignment prefixes.

The rest of the paper is organized as follows. In the next section, we briefly introduce Linda. In Sections 3 and 4 we introduce the syntax and the operational semantics of PAL, respectively. The testing theory of PAL is described in Section 5, while Section 6 contains its proof system and a small example that illustrates how the proof system works. In Section 7, by relying on standard algebraic semantics techniques, we define a denotational semantics for PAL which is fully abstract with respect to the testing preorders. In Section 8, we define syntax and operational semantics of IPAL, and show that all results for PAL smoothly generalize to the new formalism. In the last section, related work and future research are discussed.

2. A brief presentation of Linda

Linda [24,12] is a coordination language that relies on an asynchronous and associative communication mechanism based on a shared global environment called

tuple space (TS), a multiset of tuples. A *tuple* is an ordered sequence of *actual fields* (value objects) and *formal fields* (variables); the first field is always an actual field and is usually referred to as *logic* name or *tag*.

The basic interaction mechanism is *pattern-matching*; it is used to select tuples in TS. Matching is an indivisible action that permits non-deterministically selecting one of those tuples in TS with the same tag and the same number of fields as a given tuple t and such that corresponding fields have matching values or variables. Variables match any value of the same type and two values match only if identical.

There are four operations for manipulating tuple spaces: two, possibly blocking, operations for accessing and removing tuples and two non-blocking operations for adding tuples.

- **in**(t) triggers the evaluation of t and the search for a tuple t' in TS that matches t . If and when t' is found, it is removed from TS; the corresponding values of t' are assigned to the variables of t and the process continues. If no matching tuple is found, the process is suspended until one is available.
- **read**(t) is similar to **in**(t), but it does not require removal of the matched tuple t' from TS.
- **out**(t) triggers the evaluation of t and adds the outcoming tuple to TS.
- **eval**(t) is similar to **out**(t), but rather than forcing evaluation of t , it creates a new process that will evaluate t and eventually add the resulting tuple to TS.

It is worth noting that non-determinism is inherent in the definition of Linda primitives. It arises in two cases:

- different **in/read** operations are suspended waiting for the same tuple and such a tuple becomes available: only one of the suspended operations is non-deterministically selected to proceed;
- an **in/read** operation has more than one matching tuple: one is arbitrarily chosen.

The following example, borrowed from [12], is a simple C-Linda [44] solution of the dining philosophers problem. In the example, *Num* represents the number of philosophers and % the remainder of integer division.

phil(i)	initialize()
int i ;	{
{	int i ;
while (1) {	for ($i = 0$; $i < Num$; $i++$) {
think();	out ("chopstick", i);
in ("room ticket");	eval (phil(i));
in ("chopstick", i);	if ($i < (Num - 1)$) out ("room ticket");
in ("chopstick", $(i + 1) \% Num$);	}
eat();	}
out ("chopstick", i);	
out ("chopstick", $(i + 1) \% Num$);	
out ("room ticket");	
}	
}	

The use of actual fields in the argument tuple of an **in/read** instruction is known as “structured naming”. It makes TS content-addressable, in the sense that processes may select from a collection of tuples by matching the value of the component fields. Formal fields of tuples already in the tuple space are never updated, even when those tuples are used for matching **in/read** operations.

The Linda model is known as *Generative Communication* [24]. Indeed, once a tuple is added to TS (generated), its lifetime is independent from that of the producer process. This permits writing programs where complex data structures are distributed to allow different programs to work simultaneously on their elements.

3. Syntax of PAL

Since we are mainly interested in analyzing the concurrent features of the language, we assume that all values have the same type and allow only value expressions and boolean expressions. We assume existence of some predefined syntactic categories.

- *Exp*, the category of *value expressions*, which is ranged over by e , contains a set of variable symbols, *Var*, ranged over by x, y and z , and a non-empty countable set of value symbols, *Val*, ranged over by v .
- *BExp*, the category of *boolean expressions*, which is ranged over by be , contains the boolean values *false* (denoted by ff) and *true* (denoted by tt), and all boolean expressions obtained by using the usual boolean connectors (\wedge, \vee, \neg) and by applying the relational operators ($=, <, \leq, >, \geq$) to value expressions.
- \mathcal{X} , a countable set of *process variables*, which is ranged over by X, Y and Z .

We rely on the standard notions of *closed* expression, i.e. without variables, and of *substitution*. This is a function from *Var* to *Exp* which is almost everywhere the identity. We write $[e_1/x_1, \dots, e_n/x_n]$ for the substitution σ defined by $\sigma(x_i) = e_i$ whose non-trivial domain, denoted by $bv(\sigma)$, is $\{x_1, \dots, x_n\}$. We write $\sigma[e/x]$ for denoting the substitution which is the same as σ except that x is mapped to e . We write $e[e_1/x_1, \dots, e_n/x_n]$ for denoting the expression which is obtained by simultaneously substituting each occurrence of x_i in e with e_i .

We let *Tpl*, ranged over by t , denote the set of (input and output) tuples. Tuples are sequences of fields, ranged over by f . We use “ \underline{x} ” for denoting a formal field that contains the variable x , “ e ” for denoting an actual field that contains the value expression e , and “ \star ” for denoting a field that can only match fields of the same kind. For the sake of simplicity, we shall require that the variables that occur in the formal fields of each tuple be all different (this will allow us to abstract away from the evaluation ordering of tuples).

Apart for the basic Linda coordination operators (**out**, **in**, **read**, **eval**), our language has a few standard operators (see, e.g., [7,20]) for building up terms from basic ones; namely, **nil** (*inaction*), Ω (*undefined*), $a.$ (*action prefix*), **if be then - else -** (*conditional*), $[-]$ (*external choice*), \oplus (*internal choice*), $|-$ (*parallel composition*), $|-|$ (*left-merge*), $||$ (*communication-merge*) and $\text{rec}X.$ (*recursive definition*).

Variables which occur in formal fields of an input tuple t are bound by $\mathbf{in}(t)_{\dots}$ and $\mathbf{read}(t)_{\dots}$. If E is a term, we let $bv(E)$ denote the set of bound variables in E and $fv(E)$ denote that of free variables in E . Substitutions of value-expressions for variables are also extended to terms. If E is a term, we use $E\sigma$ to denote the term resulting from simultaneously substituting in E all free occurrences of $x \in Var$ with $\sigma(x)$.

The notions of free and bound process variables are the standard ones, $\mathbf{rec}X_{\dots}$ being the binding operator. Substitutions for process variables, ranged over by ξ , are mappings from process variables to terms. Their applications to terms may require renaming of bound variables for avoiding captures.

Definition 3.1. The set of *terms*, ranged over by E and F , is generated from the following grammar:

$$\begin{aligned} E &::= \mathbf{nil} \mid \Omega \mid a.E \mid \mathbf{if} \text{ be then } E_1 \text{ else } E_2 \mid E_1 \text{ op } E_2 \mid X \mid \mathbf{rec} X.E \\ a &::= \mathbf{out}(t) \mid \mathbf{in}(t) \mid \mathbf{read}(t) \mid \mathbf{eval}(E) \\ t &::= f \mid f, t \\ f &::= \underline{x} \mid e \mid \star \\ \text{op} &::= \oplus \mid \square \mid \mid \mid \sqcup \mid \mid \end{aligned}$$

We use PAL for denoting the set of all terms without free value variables and such that the variables in the formals of each tuple are all different and within the body E of subterms $\mathbf{rec} X.E$, X is not preceded by binders for value variables that are free in E . We will call *processes* those PAL terms which contain no free process variables. A process without recursion is called *finite*. We let $\mathcal{P}roc$ (ranged over by P , Q and R) denote the set of all processes.

In general, we will work with PAL terms and use E and F to range over them. Moreover, we often shall write a instead of $a.\mathbf{nil}$, and use \equiv to denote syntactical identity and Σ^{-ir} to denote the set of all the operators except for $\mathbf{in}(t)_{\dots}$ and $\mathbf{read}(t)_{\dots}$.

Like in [4], because of the interplay between process binders and value variable binders, we have to put a restriction on PAL terms. The restriction ensures that no free value variable is bound when “unfolding” $\mathbf{rec} X.E$ into $E[\mathbf{rec} X.E/X]$. Otherwise, the two terms could have different semantics. For an example, consider $P \equiv \mathbf{in}(y).\mathbf{rec} X.\mathbf{out}(y).\mathbf{in}(y).X$ and $Q \equiv \mathbf{in}(y).\mathbf{out}(y).\mathbf{in}(y).\mathbf{rec} X.\mathbf{out}(y).\mathbf{in}(y).X$. They would have different operational semantics because within P , y would be instantiated once and for all while within Q , y could be instantiated twice (actually, Q would have the same operational semantics as $\mathbf{in}(\underline{x}).\mathbf{out}(x).\mathbf{in}(y).\mathbf{rec} X.\mathbf{out}(y).\mathbf{in}(y).X$).

Notation 3.2. If t is a tuple, we let $|t|$ denote the number of fields of t (i.e. the length of t), t_j the j th field in t ($1 \leq j \leq |t|$), and $var(t)$ the set of variables in formal fields of t . With slight abuse of notation, if $t_j \equiv \underline{x}$, for some $x \in Var$, we let $var(t_j)$ denote x .

Table 1
Tuple evaluation functions

$\mathcal{O}[x] = \star$	$\mathcal{J}[x] = \underline{x}$
$\mathcal{O}[e] = \mathcal{E}[e]$	$\mathcal{J}[e] = \mathcal{E}[e]$
$\mathcal{O}[\star] = \star$	$\mathcal{J}[\star] = \star$
$\mathcal{O}[f, t] = \mathcal{O}[f], \mathcal{O}[t]$	$\mathcal{J}[f, t] = \mathcal{J}[f], \mathcal{J}[t]$

4. Operational semantics for PAL

The operational rules for our language assume existence of functions for evaluating value expressions and boolean expressions. We let them be the functions $\mathcal{E}[\cdot]:Exp \rightarrow Val$ and $\mathcal{B}[\cdot]:BExp \rightarrow \{ff, tt\}$, respectively. $\mathcal{E}[e]$ and $\mathcal{B}[be]$ will then denote the values of the expression e and of the boolean expression be provided they are closed (i.e. have no variables).

A single tuple is evaluated differently depending on whether it is an argument of **out** or of **in/read**. Tuples resulting from evaluations are elements of sets EOT and EIT . These are subsets of Tpl defined as follows.

Definition 4.1. The set of *evaluated output tuples*, EOT , ranged over by ot , and the set of *evaluated input tuples*, EIT , ranged over by it , are generated from the following grammar:

$$\begin{aligned} ot &::= of \mid of, ot & it &::= if \mid if, it \\ of &::= \star \mid v & if &::= \underline{x} \mid v \mid \star \end{aligned}$$

Since the communication capability of processes does not depend on the variables occurring in the formals of tuples, when evaluating output tuples, we do abstract away from these variables whilst, when evaluating input tuples, we need them to perform the substitutions after successful matchings. Functions $\mathcal{O}[\cdot]:Tpl \rightarrow EOT$, for output evaluations, and $\mathcal{J}[\cdot]:Tpl \rightarrow EIT$, for input evaluations, are defined inductively on the syntax of tuples in Table 1.

Pattern-matching between evaluated input and output tuples is performed by predicate *match* defined over $EIT \times EOT$ via the rules in Table 2. Our pattern-matching mechanism is slightly different from that of Linda (see Section 2). Indeed, we impose that values in input tuples can only match the same values in output tuples (hence, they cannot match formal fields of output tuples) and that the symbol \star in input tuples can only match itself, i.e. formal fields of output tuples. This separation has simplified our semantic theory; it allows us, e.g., to determine the tuple that has been accessed by an **in/read** and, also, to express **read** in terms of **in** and **out** (law READ in Table 9). We can recover the original communication capabilities of Linda input primitives by making use of \star and of the external choice operator $[\cdot]$. For example,

Table 2
Pattern-matching rules

$match(v, v)$	$v \in Val$
$match(\underline{x}, v)$	$x \in Var, v \in Val$
$match(\star, \star)$	
$\frac{match(if, of), \quad match(it, ot)}{match((if, it), (of, ot))}$	

Table 3
Action Relation (symmetrical versions of rules AR4–5 omitted)

AR1	$\mathbf{in}(t).E \xrightarrow{ot?} E[ot/\mathcal{I}[t]]$	if $match(\mathcal{I}[t], ot)$
AR2	$\mathbf{read}(t).E \xrightarrow{ot?} \mathbf{out}(ot).nil \mid E[ot/\mathcal{I}[t]]$	if $match(\mathcal{I}[t], ot)$
AR3	$\mathbf{out}(t).nil \xrightarrow{ot!} \mathbf{nil}$	
AR4	$\frac{P \xrightarrow{\alpha} P'}{P[]Q \xrightarrow{\alpha} P'}$	AR5 $\frac{P \xrightarrow{\alpha} P'}{P Q \xrightarrow{\alpha} P' Q}$
AR6	$\frac{P \xrightarrow{\alpha} P'}{P[]Q \xrightarrow{\alpha} P' Q}$	
AR7	$\frac{\mathcal{B}[be] = tt, \quad P \xrightarrow{\alpha} P'}{\mathbf{if} \text{ } be \text{ then } P \text{ else } Q \xrightarrow{\alpha} P'}$	AR8 $\frac{\mathcal{B}[be] = ff, \quad Q \xrightarrow{\alpha} Q'}{\mathbf{if} \text{ } be \text{ then } P \text{ else } Q \xrightarrow{\alpha} Q'}$

the original Linda operation $\mathbf{in}(x, v)$ matches all tuples of the form (v', v) or (v', \underline{y}) ; in PAL, $\mathbf{in}(x, v)$ “followed by” P is rendered as $\mathbf{in}(x, v).P[]\mathbf{in}(x, \star).P$.

We are, finally, ready to introduce the operational semantics of PAL.

Definition 4.2. The *operational semantics* of PAL is characterized by the extended labelled transition system $(Proc, Act, \rightarrow, \succrightarrow)$ where

- $Proc$ (i.e. the set of PAL processes) is the set of *states*,
- $Act = EOT \times \{!, ?\}$, ranged over by α , is the set of *actions* or *labels*,
- $\rightarrow \subseteq Proc \times Act \times Proc$, the *action relation*, is the least relation closed under the SOS rules in Table 3,
- $\succrightarrow \subseteq Proc \times Proc$, the *internal relation*, is the least relation closed under the SOS rules in Tables 3 and 4.

The set Act contains two kinds of actions. Action $ot!$, with $ot \in EOT$, corresponds to the production of the tuple ot because of the execution of an **out** operation. Action $ot?$, with $ot \in EOT$, corresponds to the selection of tuple ot because of the execution of an **in/read** operation. We shall use ρ to range over Act^* (i.e. sequences of actions). In rules IR12 and IR13 in Table 4, we make use of a complementation notation for labels. It is defined in the obvious way, namely $\overline{ot!} = ot?$ and $\overline{ot?} = ot!$; as usual $\overline{\overline{\alpha}} = \alpha$.

Table 4

Internal relation (symmetrical versions of rules IR7–10 omitted)

IR1	$\Omega \multimap \Omega$	IR2	$\text{rec}X.X \multimap P[\text{rec}X.X/X]$
IR3	$\frac{\mathcal{B}[be]=tt, P \multimap P'}{\text{if } be \text{ then } P \text{ else } Q \multimap P'} \quad P \neq \text{nil}$	IR4	$\frac{\mathcal{B}[be]=ff, Q \multimap Q'}{\text{if } be \text{ then } P \text{ else } Q \multimap Q'}$
IR5	$\frac{}{\text{out}(t).P \multimap \text{out}(t).\text{nil} \mid P}$	IR6	$\text{eval}(P).Q \multimap P \mid Q$
IR7	$P \oplus Q \multimap P$	IR8	$\frac{P \multimap P'}{P[\Box]Q \multimap P'[\Box]Q}$
IR9	$\frac{P \mid Q \multimap P' \mid Q}{P \multimap P'}$	IR10	$\frac{P \mid Q \multimap P' \mid Q}{P \parallel Q \multimap P' \parallel Q}$
IR11	$\frac{P \parallel Q \multimap P' \parallel Q}{P \stackrel{\alpha}{\rightarrow} P', Q \stackrel{\bar{\alpha}}{\rightarrow} Q'}$	IR13	$\frac{P \stackrel{\alpha}{\rightarrow} P', Q \stackrel{\bar{\alpha}}{\rightarrow} Q'}{P \parallel Q \multimap P' \parallel Q'}$
IR12	$\frac{P \stackrel{\alpha}{\rightarrow} P', Q \stackrel{\bar{\alpha}}{\rightarrow} Q'}{P \mid Q \multimap P' \mid Q'}$		

Most of the operational rules are similar to those for TCCS in [28] and for ACP in [1]. Commutativity and associativity of the operator for parallel composition enable us to model the actual TS as parallel composition of processes representing single tuples. The fact that TS is not modelled as a passive component allows us to represent the states of the transition system as purely syntactical objects. The asynchronous nature of the communication paradigm is rendered by allowing term P of $\text{out}(t).P$ to proceed before tuple t is actually accessed. Thus $\text{out}(t).P$ is rendered as $(\text{out}(t).\text{nil}) \mid P$ (rule IR5 in Table 4), and tuples can be used independently of what the remainders of producer processes do.

In Table 3, rule AR1 shows that process $\text{in}(t).E$ consumes a tuple ot matching the tuple $\mathcal{J}[t]$ resulting from the evaluation of t ; this causes the substitution, denoted by $E[ot/\mathcal{J}[t]]$, in E of the free occurrences of the variables in the formals of $\mathcal{J}[t]$ with the corresponding values in ot . The corresponding label carries information about the tuple consumed. Rule AR2 shows that process $\text{read}(t).E$ differs from $\text{in}(t).E$ because it leaves in TS the accessed tuple. According to the terminology of [38], the PAL operational rules adopt an *early* instantiation scheme; value variables bound by **in/read** are instantiated when input transitions are inferred, not when communications take place (*late* instantiation).

In Table 4, rule IR6 shows that **eval** causes dynamic process creation; $\text{eval}(\text{out}(t).\text{nil})$ can be used to express the original Linda $\text{eval}(t)$, that allowed tuples and not terms as arguments of **eval**. Rules AR7, AR8, IR3 and IR4 show that the conditional term **if be then P else Q** acts like P if the boolean expression be evaluates to true and like Q otherwise. Rules IR12 and IR13 deal with interprocess communication.

Rules AR4–6 and IR7–11 are similar to those for TCCS of [28] and for ACP of [1].

In the following, we shall use the summation $\sum_{i \in I}$ to represent a general *external choice operator* with $|I|$ arguments. This is justified by the SOS operational rules for \Box (AR4 and IR8, and their symmetrical ones) which imply that \Box is associative and

Table 5

Process input tuples (symmetrical versions of rules PP3–4 omitted)

PP1	$\mathbf{in}(t).E \xrightarrow{\mathcal{J}[t]}$	PP2	$\mathbf{read}(t).E \xrightarrow{\mathcal{J}[t]}$
PP3	$\frac{P \xrightarrow{it}}{P[]Q \xrightarrow{it}}$	PP4	$\frac{P \xrightarrow{it}}{P Q \xrightarrow{it}}$
PP5	$\frac{P \xrightarrow{it}}{P \parallel Q \xrightarrow{it}}$		
PP6	$\frac{\mathcal{B}[be] = tt, P \xrightarrow{it}}{\mathbf{if } be \text{ then } P \text{ else } Q \xrightarrow{it}}$	PP7	$\frac{\mathcal{B}[be] = ff, Q \xrightarrow{it}}{\mathbf{if } be \text{ then } P \text{ else } Q \xrightarrow{it}}$

commutative. By convention, $\sum_{i \in \emptyset} P_i$ denotes **nil**. Similarly, $\sum_{i \in I}$ represents a general *internal choice operator*.

4.1. Basic properties of the transition system

The LTS for PAL is in general not finitely branching since the set of initial actions that a process can perform may be infinite; processes of the form $\mathbf{in}(t).E$ can have an infinite number of derivations $\mathbf{in}(t).E \xrightarrow{ot?} E[ot/\mathcal{J}[t]]$, where $\text{match}(\mathcal{J}[t], ot)$. However, some finiteness results about the LTS can be proved, that will be useful to express the interaction ability of processes in terms of finite sets.

To single out the set of evaluated input tuples that a process can initially use for accessing tuples, we use the unary relation \xrightarrow{it} , where $it \in EIT$, defined inductively on the syntax of terms in Table 5.

Definition 4.3. For any process P and action α , we define the sets of

- *Input-read tuples*: $\text{IRT}(P) = \{it \in EIT \mid P \xrightarrow{it}\}$,
- *Output tuples*: $\text{OT}(P) = \{ot \in EOT \mid \exists P' : P \xrightarrow{ot!} P'\}$,
- *Output derivatives*: $\text{OD}(P) = \{P' \mid \exists ot \in EOT : P \xrightarrow{ot!} P'\}$,
- *Internal derivatives*: $\text{ID}(P) = \{P' \mid P \xrightarrow{} P'\}$,
- α *derivatives*: $\text{D}(P, \alpha) = \{P' \mid P \xrightarrow{\alpha} P'\}$.

Now, the following finiteness results can be established.

Proposition 4.4. For every process P and every action α , $\text{IRT}(P), \text{OT}(P), \text{OD}(P), \text{ID}(P)$ and $\text{D}(P, \alpha)$ are finite.

Proof. The proof goes by structural induction on P like in [29]. We only consider some of the most significant cases. The remaining ones are trivial or similar to those explicitly considered.

If $P \equiv \text{rec}X.E$ then we have $\text{IRT}(P) = \text{OD}(P) = \text{OT}(P) = \emptyset$, $\text{ID}(P) = \{E[\text{rec}X.E \setminus X]\}$ and $\text{D}(P, \alpha) = \emptyset$ for all $\alpha \in \mathcal{Act}$.

If $P \equiv \mathbf{out}(t).Q$ then we have the following two subcases:

- if $Q \equiv \mathbf{nil}$ then $\text{IRT}(P) = \text{ID}(P) = \emptyset$, $\text{OD}(P) = \{\mathbf{nil}\}$, $\text{OT}(P) = \{ot\}$ where $ot = \mathcal{O}[t]$ and if $\alpha = ot!$ then $\text{D}(P, \alpha) = \{\mathbf{nil}\}$ else $\text{D}(P, \alpha) = \emptyset$;
- if $Q \neq \mathbf{nil}$ then $\text{IRT}(P) = \text{OD}(P) = \text{OT}(P) = \emptyset$, $\text{ID}(P) = \{\mathbf{out}(t).\mathbf{nil}|E\}$ and $\text{D}(P, \alpha) = \emptyset$ for all $\alpha \in \mathcal{Act}$.

If $P \equiv \mathbf{in}(t).E$, with $\text{fv}(E) \subseteq \text{var}(t)$, then by definition we have $\text{IRT}(P) = \{\mathcal{I}[t]\}$, $\text{OD}(P) = \text{OT}(P) = \text{ID}(P) = \emptyset$ and if $\alpha = ot?$ then $\text{D}(P, \alpha) = \{E[ot/\mathcal{I}[t]]\}$ else $\text{D}(P, \alpha) = \emptyset$.

If $P \equiv P_1|P_2$ then we have

1. $\text{IRT}(P) = \text{IRT}(P_1) \cup \text{IRT}(P_2)$,
2. $\text{OD}(P) = \{P'_1|P_2 \mid P'_1 \in \text{OD}(P_1)\} \cup \{P_1|P'_2 \mid P'_2 \in \text{OD}(P_2)\}$,
3. $\text{OT}(P) = \text{OT}(P_1) \cup \text{OT}(P_2)$,
4. $\text{D}(P, \alpha) = \{P'_1|P_2 \mid P'_1 \in \text{D}(P_1, \alpha)\} \cup \{P_1|P'_2 \mid P'_2 \in \text{D}(P_2, \alpha)\}$,
5. $\text{ID}(P) = \{P'_1|P_2 \mid P'_1 \in \text{ID}(P_1)\} \cup \{P_1|P'_2 \mid P'_2 \in \text{ID}(P_2)\} \cup \{P'_1|P'_2 \mid (P'_1 \in \text{OD}(P_1) \vee P'_2 \in \text{OD}(P_2)) \wedge P_1 \xrightarrow{\alpha} P'_1 \wedge P_2 \xrightarrow{\bar{\alpha}} P'_2\}$.

By induction, the sets of cases 1–4 only have a finite number of elements. This is obviously true also for the sets $\{P'_1|P_2 \mid P'_1 \in \text{ID}(P_1)\}$ and $\{P_1|P'_2 \mid P'_2 \in \text{ID}(P_2)\}$ of case 5. For the set $\{P'_1|P'_2 \mid (P'_1 \in \text{OD}(P_1) \vee P'_2 \in \text{OD}(P_2)) \wedge P_1 \xrightarrow{\alpha} P'_1 \wedge P_2 \xrightarrow{\bar{\alpha}} P'_2\}$ observe that either α or $\bar{\alpha}$ must be an output action and therefore, by induction, there is only a finite number of such pairs. Let us assume that $\alpha = ot!$ and $\bar{\alpha} = ot?$ for some $ot \in \mathcal{EOT}$; by induction, we conclude that the number of possible P'_1 and P'_2 , and then that of processes of the form $P'_1|P'_2$, is finite. \square

5. Testing semantics for PAL

In this section we show how to apply the standard theory of *testing* [19,28] to PAL. To this aim we must define a set of observers, an observation mechanism (experiments and computations) and a criterion for interpreting observations. This machinery will give rise to a preorder over PAL processes formulated in terms of the inability to respond negatively to a test.

We assume a special action prefix, **success**, and a special label, ω , which are used to denote success. The operational rule which corresponds to this new prefix is: **success**. $P \xrightarrow{\omega} P$.

Observers, ranged over by O , are processes which contain the special prefix **success**.

Experiments are terms of the form $P|O$. To determine the result of an experiment $P|O$ we must consider all of its *computations*, i.e. all sequences

$$P|O \equiv P_0|O_0 \longrightarrow P_1|O_1 \longrightarrow P_2|O_2 \dots P_k|O_k \longrightarrow \dots$$

which are either infinite or such that their last pair cannot perform any internal transition. We write $P \text{ must } O$ if for each computation there exists $n \geq 0$ such that $O_n \xrightarrow{\omega}$.

We write P *must* O if P *must* O does not hold.¹

Definition 5.1. The *testing preorder* \sqsubseteq_M over PAL processes we are interested in is defined by: for all processes P and Q ,

$$P \sqsubseteq_M Q \text{ if, for every observer } O, P \text{ must } O \text{ implies } Q \text{ must } O.$$

The preorder is extended to terms which may contain free process variables as follows:

$$E \sqsubseteq_M F \text{ if, for any substitution } \xi \text{ such that } E\xi \text{ and } F\xi \text{ are processes, } E\xi \sqsubseteq_M F\xi.$$

We will use \simeq_M to denote the equivalence obtained as the kernel of the preorder (i.e. $\simeq_M = \sqsubseteq_M \cap (\sqsubseteq_M)^{-1}$).

To simplify some of the proofs, we shall introduce an alternative characterization of \sqsubseteq_M . This characterization provides an observers independent method for checking whether two processes are behaviourally related. Like in [29], this alternative characterization will rely on the *events* that processes can be engaged in and on the *sequences of actions* they can perform.

We start by introducing the notion of *patterns* of evaluated input tuples. Patterns differs from input tuples because they abstract away from the variables occurring in the formals. These variables do not affect the communication capability of processes. For instance, processes $\mathbf{in}(\underline{x}).E$ and $\mathbf{in}(\underline{y}).F$ can initially access the same tuples.

Definition 5.2. The set of *abstract evaluated input tuples* or *patterns* (*AEIT*) ranged over by p , is generated from the following grammar:

$$\begin{aligned} p &::= fp, | fp, p \\ fp &::= _ | v | \star \end{aligned}$$

Function $\wp : EIT \rightarrow AEIT$ will return the pattern of an evaluated input tuple. Essentially, \wp abstracts away from the variables in the formals of evaluated input tuples; all of them are represented as $_$. Notation 3.2 (about tuples) is extended to patterns in the obvious way. Predicate *match* is defined over $AEIT \times EOT$ by means of the rules in Table 2 where the axiom $\text{match}(\underline{x}, v)$ ($v \in Val$) is replaced by $\text{match}(_, v)$ ($v \in Val$).

We are now set to introduce the notion of event.

Definition 5.3. The set of *events*, Ev , ranged over by e , is defined as

$$Ev = \{(i, p) \mid p \in AEIT\} \cup \{(o, ot) \mid ot \in EOT\}.$$

Predicate *match* is extended to events by letting

$$\forall (i, p), (o, ot) \in Ev: \text{match}((i, p), (o, ot)) \Leftrightarrow \text{match}(p, ot).$$

¹ In general, for any given predicate \mathcal{R} , we shall write \mathcal{R} to denote that \mathcal{R} does not hold.

If $T \subseteq AEIT$ then we let $\mathcal{MT}(T) = \{ot \in EOT \mid \exists p \in T: \text{match}(p, ot)\}$, i.e. $\mathcal{MT}(T)$ is the set of evaluated output tuples that match the patterns in T . Intuitively, each event corresponds to a set of actions: (o, ot) corresponds to $ot!$ while (i, p) corresponds to any action $ot?$ with $ot \in \mathcal{MT}(\{p\})$. The set of actions corresponding to an event e may be strictly contained in that corresponding to an event e' . For example, the set of actions corresponding to $(i, (_, 7))$ contains that corresponding to $(i, (5, 7))$. Therefore, for comparing two finite sets of events we must generalize the usual (set inclusion) relation.

Notation 5.4. If $A \subseteq Ev$ we shall use the following notations:

$$\begin{aligned} Ev_{out}(A) &= \{(o, ot) \mid (o, ot) \in A\}, & \mathcal{T}(Ev_{out}(A)) &= \{ot \mid (o, ot) \in Ev_{out}(A)\}, \\ Ev_{ir}(A) &= \{(i, p) \mid (i, p) \in A\}, & \mathcal{P}(Ev_{ir}(A)) &= \{p \mid (i, p) \in Ev_{ir}(A)\}. \end{aligned}$$

Definition 5.5. For all finite sets of events A and B we write $A \triangleleft B$ if

$$Ev_{out}(A) \subseteq Ev_{out}(B) \quad \text{and} \quad \mathcal{MT}(\mathcal{P}(Ev_{ir}(A))) \subseteq \mathcal{MT}(\mathcal{P}(Ev_{ir}(B))).$$

Intuitively, $A \triangleleft B$ means that any action corresponding to an event of A has a corresponding event in B : B has at least the same interaction capability as A .

Now we can fix some standard notions.

Definition 5.6.

- A process Q such that $P \xrightarrow{\alpha} Q$ or $P \xrightarrow{\alpha} Q$ is called α derivative or internal derivative of P .
- For $\rho \in \mathcal{Act}^*$ we inductively define $P_1 \xRightarrow{\rho} P_2$ by
 1. $P_1 \xRightarrow{\varepsilon} P_2$ if $P_1 \xrightarrow{*} P_2$;
 2. $P_1 \xRightarrow{\alpha \cdot \rho'} P_2$ if $\exists P'_1, P''_1 : P_1 \xrightarrow{\varepsilon} P'_1, P'_1 \xrightarrow{\alpha} P''_1, P''_1 \xRightarrow{\rho'} P_2$.
 Often, we shall write \Rightarrow instead of $\xRightarrow{\varepsilon}$.
- For $\rho \in \mathcal{Act}^*$ we inductively define $\downarrow \rho$ by
 1. $P \downarrow \varepsilon$ if there is no infinite computation $P \xrightarrow{*} P_1 \xrightarrow{*} P_2 \xrightarrow{*} \dots$,
 2. $P \downarrow \alpha \cdot \rho'$ if $P \downarrow \varepsilon$ and whenever $P \xRightarrow{\alpha} P'$ then $P' \downarrow \rho'$.

We write $P \uparrow \rho$ if $P \downarrow \rho$ is false.

- The language of P is $L(P) = \{\rho \in \mathcal{Act}^* \mid \exists P' : P \xRightarrow{\rho} P'\}$.
- The set of successors (or initial events) of P is

$$S(P) = \{(i, \wp(it)) \mid \exists P' : P \Rightarrow P' \wedge P' \xrightarrow{it}\} \cup \{(o, ot) \mid \exists P' : P \xRightarrow{ot!} P'\}.$$

- The acceptance set of P after $\rho \in \mathcal{Act}^*$ is: $\mathcal{A}(P, \rho) = \{S(P') \mid P \xRightarrow{\rho} P'\}$.
Acceptance sets are ordered by the preorder $\subset\subset$ defined below:

$$\mathcal{A} \subset\subset \mathcal{B} \quad \text{if for all } A \in \mathcal{A} \text{ there exists } B \in \mathcal{B} \text{ such that } B \triangleleft A.$$

The following property relies on Proposition 4.4; it can be proven by induction on $|\rho|$.

Proposition 5.7. For every process P , $P \downarrow \rho$ implies that $\mathcal{A}(P, \rho)$ is finite.

Finally, we can define the alternative preorder $\ll_{\mathbf{M}}$ over processes.

Definition 5.8. For processes P and Q , $P \ll_{\mathbf{M}} Q$ if for every $\rho \in \mathcal{Act}^*$

1. $Q \downarrow \rho$
2. $\mathcal{A}(Q, \rho) \subset \subset \mathcal{A}(P, \rho)$.

We can prove that $\sqsubseteq_{\mathbf{M}}$ and $\ll_{\mathbf{M}}$ coincide when referred to processes (Theorem 5.13). The structure of the actual proof is similar to the corresponding one for process algebras in [28,29]; due to the Linda communication paradigm, a few additional complications have to be faced. In the following we shall illustrate the main steps of the proof. Whenever the proof proceeds as in [28,29] we shall omit the details.

We shall use three sets of special observers for testing processes. The first one, *con*, tests for convergence, the second, *rej*, tests for the language generated and the last, *ac*, tests for the contents of acceptance sets. These tests rely on the ability of observers to determine which tuple has been selected by pattern-matching. This is made possible by the adoption of the pattern-matching mechanism defined by the rules in Table 2, that slightly differs from the original Linda one.

We let *succ* denote the observer **success.nil** and *isucc* the observer **eval(nil)**. *succ*. The only difference is that the former immediately succeeds whilst the latter must perform an internal transition before succeeding.

Definition 5.9. For each $\rho \in \mathcal{Act}^*$, $\alpha \in \mathcal{Act}$ and A finite subset of Ev , let the observers $con(\rho)$, $rej(\rho, \alpha)$ and $ac(\rho, A)$ be defined by

1. $con(\varepsilon) = isucc$,
 $con(ot! \cdot \rho') = isucc[\mathbf{in}(ot).con(\rho')]$,
 $con(ot? \cdot \rho') = isucc[\mathbf{(out}(ot).\mathbf{nil} \parallel con(\rho'))]$;
2. $rej(\varepsilon, ot!) = isucc[\mathbf{in}(ot).\mathbf{nil}]$,
 $rej(\varepsilon, ot?) = isucc[\mathbf{(out}(ot).\mathbf{nil})]$,
 $rej(ot! \cdot \rho', \alpha) = isucc[\mathbf{in}(ot).rej(\rho', \alpha)]$,
 $rej(ot? \cdot \rho', \alpha) = isucc[\mathbf{(out}(ot).\mathbf{nil} \parallel rej(\rho', \alpha))]$;
3. $ac(\varepsilon, A) = \sum_{e \in A} ac(e)$,
 $ac(ot! \cdot \rho', A) = isucc[\mathbf{in}(ot).ac(\rho', A)]$,
 $ac(ot? \cdot \rho', A) = isucc[\mathbf{(out}(ot).\mathbf{nil} \parallel ac(\rho', A))]$,
 where for every $e \in Ev$ the observer $ac(e)$ is defined by:
 $ac(o, ot) = \mathbf{in}(ot).succ$,
 $ac(i, p) = \mathbf{(out}(ot').\mathbf{nil} \parallel succ)$

where

$$ot'_j = \begin{cases} v_1 & \text{if } p_j = _ , \text{ where } v_1 \in Val \text{ is an arbitrarily chosen value,} \\ p_j & \text{otherwise.} \end{cases}$$

The above observers are used for proving the following three propositions

Proposition 5.10. *For every process P , $P \text{ must con}(\rho)$ if and only if $P \downarrow \rho$.*

Proposition 5.11. *For every process P , if $P \downarrow \rho$ then $P \text{ must rej}(\rho, \alpha)$ if and only if $\rho \cdot \alpha \notin L(P)$*

Proposition 5.12. *For every process P , if $P \downarrow \rho$ then $P \text{ must ac}(\rho, B)$ if and only if for all $A \in \mathcal{A}(P, \rho)$ at least one of the following two conditions holds:*

- $Ev_{out}(A) \cap Ev_{out}(B) \neq \emptyset$,
- $\exists e \in Ev_{ir}(A), \exists e' \in Ev_{ir}(B), \exists e'' \in Ev : match(e, e'') \wedge match(e', e'')$.

Theorem 5.13. *For all processes P and Q , $P \sqsubseteq_M Q$ if and only if $P \ll_M Q$.*

Proof. We start by showing that \sqsubseteq_M implies \ll_M . Let us assume that $P \sqsubseteq_M Q$ and that $P \downarrow \rho$. By Proposition 5.10, this implies that $P \text{ must con}(\rho)$. Then, by hypothesis, $Q \text{ must con}(\rho)$ and, by Proposition 5.10 again, $Q \downarrow \rho$. Let $A \in \mathcal{A}(Q, \rho)$ (obviously, if $\mathcal{A}(Q, \rho) = \emptyset$ then we have finished). This means that $\rho \in L(Q)$. If $\rho = \varepsilon$, since $\varepsilon \in L(R)$ for each process R , it obviously follows that $\mathcal{A}(P, \varepsilon) \neq \emptyset$. Otherwise, let $\rho = \rho' \cdot \alpha$. By Proposition 5.11, it follows that $Q \text{ must rej}(\rho', \alpha)$. Then, by hypothesis, $P \text{ must rej}(\rho', \alpha)$ and, by Proposition 5.11 again, we conclude that $\mathcal{A}(P, \rho) \neq \emptyset$. Moreover, from Proposition 5.7 it follows that $\mathcal{A}(P, \rho)$ is finite, say $\mathcal{A}(P, \rho) = \{B_1, B_2, \dots, B_n\}$. We must show that there exists j : $1 \leq j \leq n$ such that $B_j \triangleleft A$. We derive a contradiction from the assumption that such an index j does not exist. Indeed, for each j : $1 \leq j \leq n$ there must be an event $e_j \in B_j$ and a corresponding action that does not correspond to any event of A . For all j : $1 \leq j \leq n$, let $e'_j \in Ev$ such that one of the following conditions holds:

- $e'_j = (i, ot)$, if $\exists (i, p') \in Ev_{ir}(B_j) : match(p', ot)$ and $\forall (i, p'') \in v_{ir}(A) : match(p'', ot)$ is false,
- $e'_j = (o, ot)$, if $(o, ot) \in Ev_{out}(B_j) \setminus Ev_{out}(A)$

Let B be the set of all such e'_j . By Proposition 5.12, we get $P \text{ must ac}(\rho, B)$. By construction, $Q \text{ must ac}(\rho, B)$ because of the unsuccessful computation $Q|ac(\rho, B) \xrightarrow{*} R|ac(\varepsilon, B)$ where R is such that $Q \xrightarrow{B} R$ and $S(R) \subseteq A$. Therefore, we come to a contradiction with the hypothesis $P \sqsubseteq_M Q$ and then there must exist j : $1 \leq j \leq n$ such that $B_j \triangleleft A$, which implies the thesis.

Now, we show the converse, i.e. \ll_M implies \sqsubseteq_M . Let O be any observer such that $Q \text{ must } O$: we will show that $P \text{ must } O$ as well. There are various reasons for $Q \text{ must } O$, but the basic one is the existence of a finite unsuccessful computation $Q|O \Rightarrow Q'|O'$

(i.e. Q' and O' cannot interact) such that for some sequence ρ of visible actions $Q \xrightarrow{\rho} Q'$, $O \xrightarrow{\rho} O'$ and none of the observers in the derivation $O \xrightarrow{\rho} O'$ is able to perform an ω derivative. If $P \uparrow \rho$ then there exists a subsequence ρ' of ρ and P' such that $P \xrightarrow{\rho'} P'$ and $P' \uparrow$, and it is straightforward to obtain an unsuccessful computation for $P|O$. If $P \downarrow \rho$, the hypothesis $P \ll_{\mathbf{M}} Q$ implies that there exists a process P' such that $P \xrightarrow{\rho} P'$, $\text{ID}(P') = \emptyset$ and $S(P') \triangleleft S(Q')$. Since we Q' and O' cannot interact and $S(P') \triangleleft S(Q')$, then P' and O' cannot interact too. Therefore, the derivations $P \xrightarrow{\rho} P'$ and $O \xrightarrow{\rho} O'$ can be combined to form an unsuccessful computation $P|O \Rightarrow P'|O'$. Hence, $P \text{ must } O$. \square

Using the above alternative characterization of $\sqsubseteq_{\mathbf{M}}$ it is now easy to show that the behavioural preorder is a precongruence over Σ^{-ir} , the set of all the PAL operators but **in**(t). \dots and **read**(t). \dots

Proposition 5.14. *The operators in Σ^{-ir} preserve $\sqsubseteq_{\mathbf{M}}$.*

Proof. The proof can be done by an exhaustive case analysis. When the parallel operators $|-$, $|-|$, and $|-|$ are considered $\sqsubseteq_{\mathbf{M}}$ is used, in the all remaining cases we rely on $\ll_{\mathbf{M}}$. \square

For the operators **in**(t). \dots and **read**(t). \dots , we must take into account all the substitutions they may give rise to.

Proposition 5.15. *If for each tuple $ot \in EOT$ such that $\text{match}(\mathcal{J}[t], ot)$ it holds that $E[ot/\mathcal{J}[t]] \sqsubseteq_{\mathbf{M}} F[ot/\mathcal{J}[t]]$ then **in**(t). $E \sqsubseteq_{\mathbf{M}}$ **in**(t). F and **read**(t). $E \sqsubseteq_{\mathbf{M}}$ **read**(t). F .*

Proof. Also here we take advantage of the alternative characterization $\ll_{\mathbf{M}}$. We prove the claim for **in**(t). \dots ; the proof for **read**(t). \dots being similar. If $\rho = \varepsilon$ we have **in**(t). $E \downarrow \varepsilon$ and **in**(t). $F \downarrow \varepsilon$. We also have that $\mathcal{A}(\text{in}(t).E, \varepsilon) = \{\{(i, p)\}\} = \mathcal{A}(\text{in}(t).F, \varepsilon)$, where $p = \wp(\mathcal{J}[t])$. Every non-empty sequence of actions from **in**(t). E or **in**(t). F is of the form $ot'! \cdot \rho$, for some $ot' \in EOT$ such that $\text{match}(p, ot')$ and ρ sequence from $E[ot'/\mathcal{J}[t]]$ or $F[ot'/\mathcal{J}[t]]$. Now, we can use the hypothesis $E[ot'/\mathcal{J}[t]] \ll_{\mathbf{M}} F[ot'/\mathcal{J}[t]]$ for comparing acceptance sets and convergence, and the thesis follows. \square

6. A proof system for PAL

In this section we define a proof system for PAL processes and prove that it is sound and complete with respect to the behavioural preorder $\sqsubseteq_{\mathbf{M}}$. The proof system, that we call \mathcal{CP} , is based on a set of equational laws plus two induction rules: one handling recursively defined processes and the other dealing with input prefixes. The axioms and the inference rules of the proof system are shown in Tables 6–10. Each equation $X = Y$ has to be read as standing for the pairs of inequations $X \sqsubseteq Y$ and

Table 6

Inequations for sequential, non-deterministic processes

IC1	$X \oplus (Y \oplus Z) = (X \oplus Y) \oplus Z$
IC2	$X \oplus Y = Y \oplus X$
IC3	$X \oplus X = X$
IC4	$X \oplus Y \sqsubseteq X$
EC1	$X[](Y[]Z) = (X[]Y)[]Z$
EC2	$X[]Y = Y[]X$
EC3	$X[]X = X$
EC4	$X[]\mathbf{nil} = X$
MIX1	$X \oplus (Y[]Z) = (X \oplus Y)[](X \oplus Z)$
MIX2	$X[](Y \oplus Z) = (X[]Y) \oplus (X[]Z)$

Table 7

Inequations for Ω

UND1	$\Omega \sqsubseteq X$	UND4	$\Omega X \sqsubseteq \Omega$
UND2	$X[]\Omega \sqsubseteq \Omega$	UND5	$\Omega\ X \sqsubseteq \Omega$
UND3	$X \Omega \sqsubseteq \Omega$	UND6	$\Omega\ X \sqsubseteq \Omega$

$Y \sqsubseteq X$. We shall write $E_1 \sqsubseteq_{\mathcal{CP}} E_2$ ($E_1 =_{\mathcal{CP}} E_2$) to indicate that $E_1 \sqsubseteq E_2$ ($E_1 = E_2$) can be derived within \mathcal{CP} .

Table 6 contains the standard inequations for testing from [20,28].

The laws in Table 7 state that process Ω is less defined than every PAL process (UND1) and assert the strictness of all binary operators (strictness of \oplus follows from IC4).

The laws in Table 8 are essentially concerned with the PAL parallel operators, and show that parallel operators, when applied to finite terms, can be replaced by more primitive ones, namely \mathbf{nil} , Ω , $-[]-$, $-\oplus-$, $\mathbf{in}(t)_{-}$ and $\mathbf{out}(t).\mathbf{nil}_{-}$. The left merge operator deserves specific attention because it cannot be completely replaced; its simpler form $\mathbf{out}(t).\mathbf{nil}_{-}$ is needed as a blocking output prefix. PAR1 and PAR2 are taken from [28]. PAR3 is a modification of the interleaving law of [28] to take into account the communication paradigm used in PAL. Like the interleaving law in [1], PAR3 is a weaker version of the standard ACP axiom $X|Y = X\|Y + Y\|X + X\|Y$ which applies to *stable* processes only, i.e. processes without initial internal transitions.² Similarly, CM1 is a weaker version of a corresponding ACP axiom; it holds only if Z is stable.³ The remaining laws are obvious adaptations of similar laws of [7,8,1].

² The general ACP law is not sound with respect to \simeq_M . For instance, if we take $P' = P|Q$ and $Q' = P\|Q[]Q\|P[]P\|Q$ where $P = \mathbf{in}(5).\mathbf{out}(3).\mathbf{nil}$ and $Q = \mathbf{out}(5).\mathbf{nil} \oplus \mathbf{out}(7).\mathbf{nil}$, by letting $O = (\mathbf{in}(3).\mathbf{success}.\mathbf{nil})[](\mathbf{in}(7).\mathbf{success}.\mathbf{nil})$ we get that P' must O and Q' must O (indeed, Q' can internally become the process $\mathbf{in}(5).(\mathbf{out}(3).\mathbf{nil}|Q)[]\mathbf{out}(5).\mathbf{nil}\|P[](\mathbf{out}(7).\mathbf{nil}\|P)$ which cannot satisfy O).

³ The condition on the syntactic structure of Z is necessary for the soundness of the law. For instance, if we take $P = \mathbf{out}(5).\mathbf{nil}$, $Q = \mathbf{out}(7).\mathbf{nil}$, and $R = \mathbf{in}(5).\mathbf{out}(3).\mathbf{nil} \oplus \mathbf{in}(7).\mathbf{out}(3).\mathbf{nil}$ then, by letting $O = \mathbf{in}(3).\mathbf{success}.\mathbf{nil}$, we get that $(P[]Q)\|R$ must O and $(P\|R)[]Q\|R$ must O .

Table 8

Laws for the parallel operators

PAR1	$(X \oplus Y) Z = (X Z) \oplus (Y Z)$
PAR2	$X (Y \oplus Z) = (X Y) \oplus (X Z)$
PAR3	Let $X = \sum_{i \in I} \mathbf{in}(it_i).X_i \square \sum_{k \in K} \mathbf{out}(ot_k).\mathbf{nil} \parallel X_k$ and $Y = \sum_{j \in J} \mathbf{in}(it_j).Y_j \square \sum_{l \in L} \mathbf{out}(ot_l).\mathbf{nil} \parallel Y_l$. Let $\text{COMM}(X, Y) = \{(i, l) \text{match}(it_i, ot_l)\} \cup \{(j, k) \text{match}(it_j, ot_k)\}$; then: $X Y = ((X \parallel Y) \square (Y \parallel X) \square (X \parallel Y)) \oplus (X \parallel Y)$ if $\text{COMM}(X, Y) \neq \emptyset$ $X Y = (X \parallel Y) \square (Y \parallel X)$ if $\text{COMM}(X, Y) = \emptyset$
LM1	$(X \square Y) \parallel Z = X \parallel Z \square Y \parallel Z$
LM2	$(X \oplus Y) \parallel Z = X \parallel Z \oplus Y \parallel Z$
LM3	$\mathbf{nil} \parallel X = \mathbf{nil}$
LM4	$X \parallel \mathbf{nil} = X$
LM5	$(\mathbf{in}(it).X) \parallel Y = \mathbf{in}(it).(X Y)$ if $\text{var}(it)$ not free in Y
LM6	$(\mathbf{out}(ot).\mathbf{nil} \parallel X) \parallel Y = \mathbf{out}(ot).\mathbf{nil} \parallel (X Y)$
CM1	$(X \square Y) \parallel Z = (X \parallel Z) \square (Y \parallel Z)$ if $Z = \sum_{i \in I} \mathbf{in}(it_i).Z_i \square \sum_{k \in K} \mathbf{out}(ot_k).\mathbf{nil} \parallel X_k$
CM2	$(X \oplus Y) \parallel Z = (X \parallel Z) \oplus (Y \parallel Z)$
CM3	$X \parallel Y = Y \parallel X$
CM4	$\mathbf{nil} \parallel X = \mathbf{nil}$
CM5	$(\mathbf{in}(it).X) \parallel (\mathbf{out}(ot).\mathbf{nil} \parallel Y) = X[ot/it]Y$ if $\text{match}(it, ot)$
CM6	$(\mathbf{in}(it).X) \parallel (\mathbf{out}(ot).\mathbf{nil} \parallel Y) = \mathbf{nil}$ if $\neg \text{match}(it, ot)$
CM7	$(\mathbf{in}(it_1).X) \parallel (\mathbf{in}(it_2).Y) = \mathbf{nil}$.
CM8	$(\mathbf{out}(ot_1).\mathbf{nil} \parallel X) \parallel (\mathbf{out}(ot_2).\mathbf{nil} \parallel Y) = \mathbf{nil}$

Table 9

Linda laws

OUT1	$\mathbf{out}(ot).X = \mathbf{out}(ot).\mathbf{nil} \mid X$
OUT2	$\mathbf{out}(ot).\mathbf{nil} \parallel X \square \mathbf{out}(ot).\mathbf{nil} \parallel Y = \mathbf{out}(ot).\mathbf{nil} \parallel (X \oplus Y)$
OUT3	$\mathbf{out}(ot).\mathbf{nil} \parallel X \oplus \mathbf{out}(ot).\mathbf{nil} \parallel Y = \mathbf{out}(ot).\mathbf{nil} \parallel (X \oplus Y)$
EVAL	$\mathbf{eval}(X).Y = X Y$
READ	$\frac{it \leftrightarrow t}{\mathbf{read}(it).X = \mathbf{in}(it).(\mathbf{out}(t).\mathbf{nil} \mid X)}$
IN1	$\frac{it \leftrightarrow it'}{\mathbf{in}(it).X \square \mathbf{in}(it').Y = \mathbf{in}(it).\mathbf{if} \text{ bem}(it, it') \text{ then } X \oplus Y \text{ else } X}$
IN2	$\frac{it \leftrightarrow it', pf(it) \cap pa(it') \neq \emptyset, pf(it') \cap pa(it) \neq \emptyset,}{\mathbf{in}(it).X \square \mathbf{in}(it').Y = \mathbf{in}(it).\mathbf{if} \text{ bem}(it, it') \text{ then } X \oplus (Y\sigma(it, it')) \text{ else } X \square \mathbf{in}(it').\mathbf{if} \text{ bem}(it', it) \text{ then } X\sigma(it', it) \oplus Y \text{ else } Y}$
IN3	$\frac{it \leftrightarrow it'}{\mathbf{in}(it).X \oplus \mathbf{in}(it').Y = \mathbf{in}(it).\mathbf{if} \text{ bem}(it, it') \text{ then } X \oplus (Y\sigma(it, it')) \text{ else } X \oplus \mathbf{in}(it').\mathbf{if} \text{ bem}(it', it) \text{ then } X\sigma(it', it) \oplus Y \text{ else } Y}$

The laws in Table 9 are almost all new and depend on the communication paradigm of the language. They rely on the following notations.

Notation 6.1. For any $it, it' \in EIT$ and $t \in Tpl$ such that $|it| = |it'| = |t|$:

1. $pf(it)$ denotes the set of positions of formal fields in it and $pa(it)$ the set of positions of actual fields in it ;

2. $it \leftrightarrow t$ if for each $j: 1 \leq j \leq |it|$, $j \notin pf(it)$ implies $t_j \equiv it_j$ and $j \in pf(it)$ implies $t_j \equiv var(it_j)$;
3. $it \rightleftharpoons it'$ if for each $j: 1 \leq j \leq |it|$, $j \notin pf(it) \cap pa(it')$ implies $it_j \equiv it'_j$;
4. $it \rightleftharpoons it'$ if for each $j: 1 \leq j \leq |it|$, $j \notin (pf(it) \cap pa(it')) \cup (pf(it') \cap pa(it))$ implies $it_j \equiv it'_j$;
5. $bem(it, it') = (\bigwedge_{j \in (pf(it) \cap pa(it'))} var(it_j) = it'_j)$;
6. $\sigma(it, it') = [it_j / var(it'_j)]_{j \in (pf(it') \cap pa(it))}$.

Let us comment on the notations introduced above.

1. As an example, consider the tuple $(\underline{x}, 3, \underline{y}, \star)$. Then, we have $pf((\underline{x}, 3, \underline{y}, \star)) = \{1, 3\}$ and $pa((\underline{x}, 3, \underline{y}, \star)) = \{2\}$.
2. $it \leftrightarrow t$ means that t can be obtained from it by removing the line under the variables in the formals, hence by transforming formals into actuals. For example, $(\underline{x}, 7, \star) \leftrightarrow (x, 7, \star)$.
3. $it \rightleftharpoons it'$ states that all the output tuples that match it' do match it as well and that the variables occurring in the formals of it' also occur in the corresponding formals of it . For example, $(\underline{x}, \star, \underline{y}) \rightleftharpoons (\underline{x}, \star, 7)$, but $(\underline{x}, \star, \underline{y}) \rightleftharpoons (\underline{z}, \star, 7)$ and $(\underline{x}, \star, \underline{y}) \rightleftharpoons (\underline{x}, \underline{z}, 7)$ do not hold.
4. $it \rightleftharpoons it'$ states that the corresponding formals in it and it' are syntactically identical and that there exist output tuples that can match both it and it' . For example, $(\underline{x}, \star, 7) \rightleftharpoons (5, \star, \underline{y})$. Note that $it \rightleftharpoons it'$ implies $it \rightleftharpoons it'$.
5. $bem(it, it')$ can be understood as a boolean function that evaluates to true whenever it receives as argument an output tuple that matches both it and it' , and $it \rightleftharpoons it'$ or $it \rightleftharpoons it'$. In general, $bem(it, it')$ has the form $x_1 = v_1 \bigwedge \dots \bigwedge x_n = v_n$ with $x_j \in var(it)$ and $v_j \in Val$ for $1 \leq j \leq n$. We will always use $bem(it, it')$ under the scope of the operator $\mathbf{in}(it)_{\dots}$, that will bind subsequent free occurrences of variables in $var(it)$. As usual, we let $\bigwedge_{j \in \emptyset} be_j = tt$.
6. $\sigma(it, it')$ is a substitution that replaces the variables occurring in the formals of it' with the corresponding values of it .

The assumption that the variables occurring in the formals of each tuple be all different has been important for having simple definitions of $bem(it, it')$ and of $\sigma(it, it')$.

Let us comment on the laws in Table 9. Laws OUT1 and EVAL assert that both $\mathbf{out}(ot)_{\dots}$ and $\mathbf{eval}(E)_{\dots}$ are non blocking operators. In particular, OUT1 says that our general output prefixing is not needed; nullary process operators of the forms $\mathbf{out}(ot)_{\dots}\mathbf{nil}$ are sufficient. Laws OUT2 and OUT3 make evident the internal non-determinism of processes and permit postponing internal choices.

READ permits expressing the operator **read** in terms of **in** and **out**. The law relies on the ability of determining which tuple has been selected by pattern-matching. Note that, from the definition of $it \leftrightarrow t$, it follows that $fv(\mathbf{out}(t)_{\dots}\mathbf{nil}) \subseteq var(it)$. As a simple application of the law we get the equation $\mathbf{read}(\star, \underline{x})_{\dots}\mathbf{nil} =_{\mathcal{P}} \mathbf{in}(\star, \underline{x})_{\dots}(\mathbf{out}(\star, x)_{\dots}\mathbf{nil}|\mathbf{nil})$.

IN1 permits deleting the second summand of an external choice by absorbing its behaviour in that of the first. The premise guarantees that the law is applied only if the choice is internal, i.e. the first summand may access all the tuples accessible by

the second. For instance, since $\underline{x} \Leftarrow 2$ and $bem((\underline{x}), (2)) = [x = 2]$, from IN1 we get the equation

$$\begin{aligned} & \mathbf{in}(\underline{x}).\mathbf{out}(1).\mathbf{nil}[]\mathbf{in}(2).\mathbf{nil} \\ &=_{\mathcal{CP}} \mathbf{in}(\underline{x}).\mathbf{if } x = 2 \mathbf{ then } (\mathbf{out}(1).\mathbf{nil} \oplus \mathbf{nil}) \mathbf{ else out}(1).\mathbf{nil}. \end{aligned}$$

IN2 permits introducing an internal choice after a tuple has been accessed. The premises guarantee that the law is applied only if the summands of an external choice can access common tuples ($it \Leftarrow it'$) and it is not the case that a summand may access all the tuples the other summand may access ($pf(it) \cap pa(it') \neq \emptyset$ and $pf(it') \cap pa(it) \neq \emptyset$). The law says that for both summands it is possible to access a common tuple without making any commitment about the next behaviour. The premise $it \Leftarrow it'$, the definition of $\sigma(it, it')$, and the fact that $fv(X) \subseteq var(it)$ and $fv(Y) \subseteq var(it')$ imply that $fv(X\sigma(it', it)) \subseteq var(it')$ and $fv(Y\sigma(it, it')) \subseteq var(it)$. For an example, consider process

$$Q \equiv \mathbf{in}(\underline{x}, \underline{z}, 5).\mathbf{out}(x, z).\mathbf{nil}[]\mathbf{in}(7, \underline{z}, \underline{y}).\mathbf{out}(y, z).\mathbf{nil}.$$

Since we have

$$\begin{aligned} & (\underline{x}, \underline{z}, 5) \Leftarrow (7, \underline{z}, \underline{y}), \quad pf((\underline{x}, \underline{z}, 5)) \cap pa((7, \underline{z}, \underline{y})) = \{1\}, \\ & pf((7, \underline{z}, \underline{y})) \cap pa((\underline{x}, \underline{z}, 5)) = \{3\}, \\ & bem((\underline{x}, \underline{z}, 5), (7, \underline{z}, \underline{y})) = [x = 7], \quad bem((7, \underline{z}, \underline{y}), (\underline{x}, \underline{z}, 5)) = [y = 5], \\ & \sigma((\underline{x}, \underline{z}, 5), (7, \underline{z}, \underline{y})) = [5/y] \text{ and } \sigma((7, \underline{z}, \underline{y}), (\underline{x}, \underline{z}, 5)) = [7/x], \end{aligned}$$

it follows that from IN2 we can derive the equation

$$\begin{aligned} Q &=_{\mathcal{CP}} \mathbf{in}(\underline{x}, \underline{z}, 5).\mathbf{if } x = 7 \mathbf{ then out}(x, z).\mathbf{nil} \oplus \mathbf{out}(5, z).\mathbf{nil} \mathbf{ else out}(x, z).\mathbf{nil} \\ & \quad [] \mathbf{in}(7, \underline{z}, \underline{y}).\mathbf{if } y = 5 \mathbf{ then out}(7, z).\mathbf{nil} \oplus \mathbf{out}(y, z).\mathbf{nil} \mathbf{ else out}(y, z).\mathbf{nil}. \end{aligned}$$

Laws IN1 and IN2 are mutually exclusive, in the sense that if one can be applied the other cannot.

IN3 rests on the same ideas of IN1 and IN2. No summand is absorbed and law IN3 makes it evident the internal nondeterminism due to the fact that there exist output tuples that both summands can access. IN1 and IN3 allow us to derive for in laws similar to OUT2 and OUT3 (see D1 and D2 in Table 11).

The rules of the proof system are in Table 10. Most of them are borrowed from [28] and should be self-explanatory. The main addition is III (a similar rule was already present in [29]). It is infinitary if Val , hence EOT , is infinite. Thus, our proof system has two infinitary rules: VI for handling recursively defined terms and III for dealing with input prefixes. The use of infinitary rules makes the completeness result of purely theoretical interest. In practice, more tractable forms of induction are needed (one of these forms shall be used in the example presented at the end of this section). In VI, we use E^n to denote the n th finite syntactic approximant of E . This is a standard construction of algebraic semantics and the actual definition can be found in, e.g., [28]. The basic idea is that every term E determines a set of finite terms (i.e. without recursion) that are obtained by unfolding a finite number of times the recursive (sub)terms. In $V(a)$, ξ ranges over substitutions for process variables. VII and VIII

Table 10
Inference rules

I (a) $\frac{}{E \sqsubseteq E}$	I (b) $\frac{E_1 \sqsubseteq E_2, E_2 \sqsubseteq E_3}{E_1 \sqsubseteq E_3}$
II $\frac{E_1 \sqsubseteq E'_1, E_2 \sqsubseteq E'_2}{E_1 \text{ op } E_2 \sqsubseteq E'_1 \text{ op } E'_2} \quad \text{op} \in \{\oplus, [], , \parallel, \}$	
III $\frac{E_1[ot/it] \sqsubseteq E_2[ot/it]}{\mathbf{in}(it).E_1 \sqsubseteq \mathbf{in}(it).E_2} \quad \text{match}(it, ot)$	
IV (a) $\frac{E_1 \sqsubseteq E_2}{\text{rec}X.E_1 \sqsubseteq \text{rec}X.E_2}$	IV (b) $\frac{}{\text{rec}X.E = E[\text{rec}X.E/X]}$
V (a) $\frac{E_1 \sqsubseteq E_2}{E_1 \zeta \sqsubseteq E_2 \zeta}$	V (b) $\frac{}{E_1 \sqsubseteq E_2} \text{ for every law } E_1 \sqsubseteq E_2$
VI $\frac{\forall n \geq 0 : E_1^n \sqsubseteq E_2}{E_1 \sqsubseteq E_2}$	
VII (a) $\frac{\mathcal{B}[be] = tt}{\mathbf{if } be \text{ then } E_1 \text{ else } E_2 = E_1}$	VII (b) $\frac{\mathcal{B}[be] = ff}{\mathbf{if } be \text{ then } E_1 \text{ else } E_2 = E_2}$
VIII (a) $\frac{\mathcal{C}[t] = \mathcal{C}[t']}{\mathbf{out}(t).\mathbf{nil} \sqsubseteq E = \mathbf{out}(t').\mathbf{nil} \sqsubseteq E}$	VIII (b) $\frac{\mathcal{S}[t] = \mathcal{S}[t']}{\mathbf{in}(t).E = \mathbf{in}(t').E}$
IX $\frac{x \in \text{var}(t), y \text{ fresh}}{\mathbf{in}(t).E = \mathbf{in}(t[y/x]).E[y/x]}$	

Table 11
Derived laws

D1	$\mathbf{in}(it).X \parallel \mathbf{in}(it).Y = \mathbf{in}(it).(X \oplus Y)$
D2	$\mathbf{in}(it).X \oplus \mathbf{in}(it).Y = \mathbf{in}(it).(X \oplus Y)$
D3	$X \oplus Y = X \oplus Y \oplus (X \parallel Y)$
D4	$X \oplus (X \parallel Y \parallel Z) = X \oplus (X \parallel Y) \oplus (X \parallel Y \parallel Z)$
D5	$((\mathbf{out}(ot).\mathbf{nil} \parallel X_1) \parallel Y_1) \oplus ((\mathbf{out}(ot).\mathbf{nil} \parallel X_2) \parallel Y_2) =$ $((\mathbf{out}(ot).\mathbf{nil} \parallel (X_1 \oplus X_2)) \parallel Y_1) \oplus ((\mathbf{out}(ot).\mathbf{nil} \parallel (X_1 \oplus X_2)) \parallel Y_2)$
D6	$\frac{it_1 \rightleftharpoons it_2}{((\mathbf{in}(it_1).X_1) \parallel Y_1) \oplus ((\mathbf{in}(it_2).X_2) \parallel Y_2) =$ $((\mathbf{in}(it_1).\mathbf{if } bem(it_1, it_2) \text{ then } X_1 \oplus (X_2 \sigma(it_1, it_2)) \text{ else } X_1) \parallel Y_1)$ $\oplus ((\mathbf{in}(it_2).\mathbf{if } bem(it_2, it_1) \text{ then } X_1 \sigma(it_2, it_1) \oplus X_2 \text{ else } X_2) \parallel Y_2)$

assume existence of evaluation mechanisms for expressions, boolean expressions and tuples. IX is an α -conversion rule for input prefixed terms; substitutions are applied to tuples in the obvious way.

It could be proven that each axiom is *independent* from the other (its removal would affect the relation provable in \mathcal{CP}). For the sake of space, we will not do it but, in the completeness proof, we will point out the specific rôle of each axiom. The soundness and completeness proof proceeds in two steps: first a reduced proof system is considered and its soundness and completeness for finite processes is proven; then, the inference rules are used to establish soundness and completeness of \mathcal{CP} .

Let \mathcal{RP} be the proof system obtained from \mathcal{CP} by deleting rules IV(a) and VI in Table 10, dealing with possibly infinite terms. We shall write $E \sqsubseteq_{\mathcal{RP}} F$ to indicate that $E \sqsubseteq F$ can be derived within \mathcal{RP} .

The next theorem states soundness of \mathcal{RP} . Soundness of \mathcal{CP} will rely on partial completeness of \mathcal{RP} ; the soundness proof of \mathcal{CP} will be completed in Theorem 6.22.

Theorem 6.2. *For value-closed terms E and F , $E \sqsubseteq_{\mathcal{RP}} F$ implies $E \sqsubseteq_M F$.*

Proof. The soundness proof consists in checking that the preorder \sqsubseteq_M is preserved by the rules and that the laws are satisfied by \sqsubseteq_M . Rule I states that \sqsubseteq_M is a preorder. Soundness of II and III stating the substitutivity of \sqsubseteq_M into PAL contexts is affirmed by Propositions 5.14 and 5.15. Soundness of IV(b), VII, VIII and IX can be easily proven by using the alternative characterization \ll_M of \sqsubseteq_M . Rule V(a) is sound by definition of \sqsubseteq_M over terms which are open with respect to process variables. Therefore, soundness of the proof system for open (w.r.t. process variables) terms is an easy consequence of soundness for closed ones. Soundness of V(b) reduces to that of the axioms in \mathcal{RP} . The axioms in \mathcal{RP} can be easily proven sound by using \ll_M instead of \sqsubseteq_M . \square

Let us now concentrate on proving completeness of \mathcal{RP} for finite PAL processes. The proof rests on the existence of standard forms (see, e.g., [37]) for processes called *head normal forms* (h nfs). Similar forms were already used e.g. in [19,28,29]. Intuitively, these special forms aim at describing processes as an internal non-deterministic choice among a set of initial states. In our framework, each initial state is represented by the initial events the process can perform and their derivatives.

We start introducing the notions of *closed* and *saturated set* of events. For defining a partial order over acceptance sets, we must adapt the standard saturation procedure since our basic preorder \triangleleft , used for comparing finite sets of events, is not a partial order.

Definition 6.3. A finite subset A of Ev is *closed* if for each $e \in A$, there does not exist $e' \in A$ such that $\{e'\} \triangleleft \{e\}$. We let

$$cl(A) = \{(i, p) \in A \mid \nexists (i, p') \in A: j \notin pf(p') \cap pa(p) \text{ implies } p'_j \equiv p_j\} \cup Ev_{out}(A).$$

Proposition 6.4. \triangleleft is a partial order over the set of closed sets of events.

Proof. We must show that for all A and B closed subsets of Ev , we have that $A \triangleleft B$ and $B \triangleleft A$ if and only if $A = B$. Obviously, if $A = B$ then $A \triangleleft B$ and $B \triangleleft A$. Conversely, suppose that $A \triangleleft B$ and $B \triangleleft A$. We proceed by contradiction. Let us assume that $A \neq B$. Without loss of generality, we may assume that there exists $e \in A \setminus B$. Since $A \triangleleft B$ we get that there exists $e' \in B$ such that $\{e\} \triangleleft \{e'\}$. The hypothesis $e \notin B$ implies $e \neq e'$.

Moreover, $e' \notin A$; otherwise it would contradict the fact that A is closed. Therefore, since $e' \in B \triangleleft A$ and $B \triangleleft A$, we deduce that there exists $e'' \in A$ such that $\{e'\} \triangleleft \{e''\}$. By transitivity we get that $\{e\} \triangleleft \{e''\}$ which contradicts the fact that A is closed. Therefore, $A = B$ must hold. \square

Corollary 6.5. $cl(A)$ represents the equivalence class of A with respect to \triangleleft .

Proof. By definition, $cl(A)$ is closed. Since $cl(A) \subseteq A$, by definition, we have $cl(A) \triangleleft A$. Moreover, for any $e \in A$ we have that either $e \in cl(A)$ or there exists $e' \in cl(A)$ such that $\{e\} \triangleleft \{e'\}$. Hence, $A \triangleleft cl(A)$. Therefore, from Proposition 6.4, it follows that if B is a finite subset of Ev such that $A \triangleleft B$ and $B \triangleleft A$ then $cl(A) = cl(B)$. \square

We can now define the notion of saturated set, which in our framework applies to finite collections of sets of events (i.e. acceptance sets, see Section 5). If \mathcal{A} is an acceptance set then we let $Ev(\mathcal{A})$ denote the set $\bigcup \{A \mid A \in \mathcal{A}\}$ of all events in \mathcal{A} , $Ev_{in}(\mathcal{A})$ denote the set of input events in \mathcal{A} and $Ev_{out}(\mathcal{A})$ denote the set of output events in \mathcal{A} .

Definition 6.6. A finite collection \mathcal{A} of closed sets of events is *saturated*, or is a *saturated set*, if the following conditions hold:

1. $cl(Ev(\mathcal{A})) \in \mathcal{A}$;
2. $A, B \in \mathcal{A}$, C closed: $A \triangleleft C \triangleleft B$, $C \neq A$ and $C \neq B$ imply $C \notin \mathcal{A}$.

Let $sat(Ev)$, ranged over by \mathcal{A} , \mathcal{B} , etc., be the set of all saturated sets over Ev .

It is always possible to transform an acceptance set \mathcal{A} into a saturated set \mathcal{B} such that $\mathcal{A} \subset \subset \mathcal{B}$ and $\mathcal{B} \subset \subset \mathcal{A}$. We shall use the following construction. Let $sat(\mathcal{A})$ be the greatest subset of $\mathcal{U}(\mathcal{A}) = \{cl(A) \mid A \in \mathcal{A}\} \cup \{cl(Ev(\mathcal{A}))\}$ such that

1. $cl(Ev(\mathcal{A})) \in sat(\mathcal{A})$;
2. $A \in \mathcal{U}(\mathcal{A}) \setminus \{cl(Ev(\mathcal{A}))\}$ and $\nexists B \in \mathcal{U}(\mathcal{A}) \setminus \{A\} : B \triangleleft A$ imply $A \in sat(\mathcal{A})$.

Note that for saturating a given collection of sets of events, instead of adding elements to the collection (as, e.g., in [19,28]), we delete some of the sets in the collection. This is somehow similar to the “minimization” procedure of [16] and is essential for obtaining finite collections. The saturation of acceptance sets like $\{(i, (5))\}, \{(i, (-))\}$ would otherwise result in the infinite collection whose elements are sets containing $(i, (5))$ and events of the form $(i, (v))$, with $v \in Val \setminus \{5\}$.

Proposition 6.7. For every acceptance set \mathcal{A} , $sat(\mathcal{A}) \subset \subset \mathcal{A}$ and $\mathcal{A} \subset \subset sat(\mathcal{A})$.

Proof. We prove that $sat(\mathcal{A}) \subset \subset \mathcal{A}$ by case analysis on the elements of $sat(\mathcal{A})$:

- if $A = cl(Ev(\mathcal{A}))$ then, by definition, $\forall B \in \mathcal{A} : B \triangleleft A$;
- if $A \neq cl(Ev(\mathcal{A}))$ then, by definition, $\exists B \in \mathcal{A} : A = cl(B)$; hence $B \triangleleft A$.

We now prove that $\mathcal{A} \subset \subset sat(\mathcal{A})$. Let $A \in \mathcal{A}$; then $cl(A) \in \mathcal{U}(\mathcal{A})$. We have two cases to consider. If $cl(A) \in sat(\mathcal{A})$, then $B = cl(A)$ is such that $B \triangleleft A$. Suppose now that $cl(A) \notin sat(\mathcal{A})$. Since $sat(\mathcal{A})$ is the greatest subset of $\mathcal{U}(\mathcal{A})$ which enjoys

properties 1 and 2 of Definition 6.6, then there must exist $B \in \text{sat}(\mathcal{A})$: $B \triangleleft \text{cl}(A)$ (otherwise also $\text{sat}(\mathcal{A}) \cup \{\text{cl}(A)\}$ would enjoy the previous properties); by transitivity, $B \triangleleft A$. \square

Proposition 6.8. $\subset\subset$ is a partial order over $\text{sat}(\text{Ev})$.

Proof. We must show that for \mathcal{A} and \mathcal{B} saturated sets, we have that $\mathcal{A} \subset\subset \mathcal{B}$ and $\mathcal{B} \subset\subset \mathcal{A}$ if and only if $\mathcal{A} = \mathcal{B}$. Obviously, if $\mathcal{A} = \mathcal{B}$ then $\mathcal{A} \subset\subset \mathcal{B}$ and $\mathcal{B} \subset\subset \mathcal{A}$. Conversely, suppose that $\mathcal{A} \subset\subset \mathcal{B}$ and $\mathcal{B} \subset\subset \mathcal{A}$. We proceed by contradiction. Let us assume that $\mathcal{A} \neq \mathcal{B}$. If $A = \text{cl}(\text{Ev}(\mathcal{A})) \notin \mathcal{B}$ then either $\exists A' \in \mathcal{A} \setminus \mathcal{B}$: $A' \neq A$ (if $\text{cl}(\text{Ev}(\mathcal{B})) \triangleleft \text{cl}(\text{Ev}(\mathcal{A}))$) or $\exists B \in \mathcal{B} \setminus \mathcal{A}$: $B \neq A$ and $B \neq \text{cl}(\text{Ev}(\mathcal{B}))$ (if $\text{cl}(\text{Ev}(\mathcal{A})) \triangleleft \text{cl}(\text{Ev}(\mathcal{B}))$). Therefore, without loss of generality, we may assume that there exists $A \in \mathcal{A} \setminus \mathcal{B}$ such that $A \neq \text{cl}(\text{Ev}(\mathcal{A}))$. Since $\mathcal{A} \subset\subset \mathcal{B}$, then there exists $B \in \mathcal{B}$ such that $B \triangleleft A$. Moreover, $\mathcal{B} \subset\subset \mathcal{A}$ implies that there exists $A' \in \mathcal{A}$ such that $A' \triangleleft B$. By transitivity, we have $A' \triangleleft A$. Since \mathcal{A} is saturated and $A \neq \text{cl}(\text{Ev}(\mathcal{A}))$, then $A' = A$. By Proposition 6.4, we get $A = B$ which contradicts the hypothesis that $A \notin \mathcal{B}$. \square

We now introduce our standard forms for processes.

Definition 6.9 (*Head normal forms*)

- We let \cong to denote the least equivalence relation induced by the following rules:

$$\frac{\mathcal{B}[\text{be}] = tt}{E \cong \text{if } \text{be} \text{ then } E \text{ else } F} \quad \text{and} \quad \frac{\mathcal{B}[\text{be}] = ff}{E \cong \text{if } \text{be} \text{ then } F \text{ else } E}.$$

- A partial function $g: \text{Ev} \rightarrow \mathcal{P}\text{roc}$ is a *normal function* (*nf*) if
 - (a) $(o, ot) \in \text{dom}(g)$ implies $g((o, ot)) = \text{out}(ot). \mathbf{nil} \parallel P$ for some P ;
 - (b) $(i, p) \in \text{dom}(g)$ implies $g((i, p)) = \mathbf{in}(it).E$, for some E , and $\wp(it) = p$;
 - (c) $g((i, p_1)) = \mathbf{in}(it_1).E$, $g((i, p_2)) = \mathbf{in}(it_2).F$ and $it_1 \rightleftharpoons it_2$ imply $E\sigma(it_2, it_1) \cong F\sigma(it_1, it_2)$.
- A process P is a *head normal form* (*hnf*) if one of the following conditions holds:
 - $P \equiv \sum_{e \in A} g(e)$ where A is a closed set of events, g is a *nf* and $\text{dom}(g) = A$;
 - $P \equiv \sum_{A \in \mathcal{A}} \sum_{e \in A} g(e)$ where \mathcal{A} is saturated, g is a *nf* and $\text{dom}(g) = \text{Ev}(\mathcal{A})$.

Let us comment on the definition above, where we have used a terminology introduced in Notations 6.1. Intuitively, if $P \cong Q$ then $\forall \alpha \in \mathcal{A}ct$, $P \xrightarrow{\alpha} R$ ($P \succ \rightarrow R$) if and only if $Q \xrightarrow{\alpha} R$ ($Q \succ \rightarrow R$), i.e. P and Q have the same derivatives, which, obviously imply that P and Q are equivalent. Moreover, recall that $\sum_{e \in \emptyset} g(e)$ denotes \mathbf{nil} ; hence, \mathbf{nil} is a *hnf*. From the definition it should be evident that if P is in *hnf* then $P \downarrow e$. The functional notation has been adopted for pointing out that in a *hnf* each event e is associated with a single term $g(e)$. Condition (c) ensures that for each initial action α that a *hnf* P can perform, all α derivatives of P are \cong -equivalent, hence testing equivalent. Indeed, (c) checks that whenever a tuple $ot' \in \{ot \mid \text{match}(it_1, ot), \text{match}(it_2, ot)\}$ is accessed the obtained processes are \cong -equivalent. Therefore, for a given α , we will not distinguish among α derivatives of a *hnf* P . In the sequel, we will consistently use the notation P_α for denoting α derivatives of a *hnf* P .

The counterpart of head normal forms for divergent but finite processes are Ω -head normal forms.

Definition 6.10. A process P is an Ω -head normal form, Ω -hnf for short, if one of the following conditions holds:

1. \mathbf{nil} and Ω are Ω -hnf;
2. $\sum_{A \in \mathcal{A}} \sum_{e \in A} P_e^A$ is a Ω -hnf if $\forall A \in \mathcal{A}, \forall e \in A, P_e^A$ is of the form $\mathbf{in}(it).E$ or of the form $\mathbf{out}(ot).\mathbf{nil} \parallel P$.

For proving completeness, we need a special induction parameter, namely the largest number of communications that a finite process can perform. This parameter can be defined in terms of the maximal number of visible actions the process can do during a derivation. We define the *depth* of a finite process P as $\text{depth}(P) = \max\{|\rho| \mid P \xRightarrow{\rho}\}$. Let E denote a value-open finite process. We generalize the definition of *depth* by letting $\text{depth}(E) = \max\{\text{depth}(P) \mid P \text{ value-closed instantiation of } E\}$. Since we have confined ourselves to finite terms without process variables this number is finite.

The following propositions about the existence of standard forms for processes will be used in the proof of the completeness theorem. The laws in Table 8 and OUT1 and EVAL in Table 9 are used for expressing the parallel operators in terms of the non-deterministic ones; the laws in Tables 6, 9 and 11 are crucial for obtaining saturated sets of events. All the laws in Table 11 can be derived within \mathcal{RP} . Indeed, the first two laws are easily derived from IN1 and IN3, respectively. D3 and D4 can be derived in exactly the same way as the corresponding ones in [28]. The last two laws can be obtained in a similar fashion as Der3 in [28, p. 97]; in particular, D6 is obtained if IN2 and IN3 are used instead of the laws which in [28] correspond to D1 and D2.

Proposition 6.11. *For any finite process P there exists a Ω -hnf, $\Omega(P)$ such that $P =_{\mathcal{RP}} \Omega(P)$.*

Proof. The actual proof goes by structural induction on P (in case $P \equiv P_1 | P_2$ it further relies on $\text{depth}(P_1 | P_2)$) and is omitted because it is similar to that of Lemma 3.4.3 in [35]. \square

Proposition 6.12. *For any finite process P , $P \uparrow \varepsilon$ if and only if $P =_{\mathcal{RP}} \Omega$.*

Proof. By the previous proposition, we may assume that P is a Ω -hnf. If $P \uparrow \varepsilon$ then P must be Ω , otherwise (it must be either \mathbf{nil} or of the form $\sum_{A \in \mathcal{A}} \sum_{e \in A} P_e^A$ where P_e^A is as in Definition 6.10) it cannot diverge. Conversely, if $P =_{\mathcal{RP}} \Omega$ then, since the proof system \mathcal{RP} is sound with respect to $\sqsubseteq_{\mathbf{M}}$, we get $P \simeq_{\mathbf{M}} \Omega$. In particular this means that $P \ll_{\mathbf{M}} \Omega$ and then $P \uparrow \varepsilon$. \square

To simplify the reduction of convergent processes to hnfs within \mathcal{RP} (Proposition 6.15) we introduce another special form for processes, namely *head sum form*

(*hsf*). We will show that every convergent process can be transformed into *hsf* and that every *hsf* can be transformed into *hnf*. Roughly speaking, a *hsf* is a process whose top-level operators are $-\oplus-$, $-\square-$ or one of **in**(*it*). $_{-}$ and **out**(*ot*).**nil** $_{-}$ in this order.

Definition 6.13. The set of *head sum forms*, HSF, is the least set of processes which satisfies:

1. **out**(*ot*).**nil** $\|P \in \text{BHSF}$, **in**(*it*). $E \in \text{BHSF}$ if $fv(E) \subseteq var(it)$;
2. $\sum_{e \in A} P_e \in \text{HSF}$ if $\forall e \in A: P_e \in \text{BHSF}$;
3. $\sum_{A \in \mathcal{A}} \sum_{e \in A} P_e^A \in \text{HSF}$ if $\forall A \in \mathcal{A}, \forall e \in A: P_e^A \in \text{BHSF}$.

Proposition 6.14. For any finite process P , if $P \downarrow \varepsilon$ then there exists a *hsf*, $s(P)$ such that $P =_{\mathcal{AP}} s(P)$.

Proof. The proof is omitted because it is similar to that of Proposition 3.4.6 in [35]. \square

Proposition 6.15. For any process P , if $P \downarrow \varepsilon$ then there exists a *hnf*, $h(P)$ such that $P =_{\mathcal{AP}} h(P)$.

Proof. Because of Proposition 6.14, we may assume that P is a *hsf*, say $P \equiv \sum_{A \in \mathcal{A}} \sum_{e \in A} P_e^A$ (the case $P \equiv \sum_{e \in A} P_e$ is similar). By repeated use of law IN1, we can rewrite P in a *hsf* $\sum_{B \in \mathcal{B}} \sum_{e \in B} P_e^B$ such that each $B \in \mathcal{B}$ is a closed set of events. To make P_e^B independent of B we repeatedly use laws MIX1, MIX2, OUT2, IN2, D5 and D6. The resulting *hsf* is of the form $\sum_{B \in \mathcal{B}} \sum_{e \in B} Q_e$; however, condition (c) of Definition 6.9, in general, is not satisfied. Laws IN2 and D6 (which generalize law IN3) can be repeatedly used to obtain a *hnf* $\sum_{B \in \mathcal{B}} \sum_{e \in B} Q'_e$ such that if we define $g(e) = Q'_e$ for each $e \in Ev(\mathcal{B})$ then g is a normal function. The last step consists in saturating the *hsf* $\sum_{B \in \mathcal{B}} \sum_{e \in B} g(e)$. This is the only missing requirement for *hsf* to be a *hnf*; it can be satisfied by using laws D3 and D4. \square

The relation we are going to define permits syntactical comparisons of *hnfs* at their top level. It is a bridge between the semantical relation \sqsubseteq_M and the proof-theoretic one $\sqsubseteq_{\mathcal{AP}}$. In the following we shall use $\mathcal{IA}(P)$ to denote the set of actions P can (initially) perform, i.e. $\mathcal{IA}(P) = \{\alpha \in \mathcal{Act} \mid \exists P' : P \xrightarrow{\alpha} P'\}$.

Proposition 6.16. If P and Q are *hnfs*, $P \sqsubseteq_M Q$ implies $P_\alpha \sqsubseteq_M Q_\alpha$, for each $\alpha \in \mathcal{IA}(P) \cap \mathcal{IA}(Q)$.

Proof. We have two cases to consider according to $\alpha = ot!$ or $\alpha = ot?$. In the first case, if $P_\alpha \text{ must } O$, let $O' \equiv (\mathbf{in}(ot).O)[]isucc$; then $P \text{ must } O'$. Therefore, by hypothesis, $Q \text{ must } O'$. Since $Q \mid O' \xrightarrow{*} Q_\alpha \mid O$ it follows that $Q_\alpha \text{ must } O$. The case $\alpha = ot?$ can be proven similarly but with $O' \equiv ((\mathbf{out}(ot).\mathbf{nil})[]O)[]isucc$. \square

Definition 6.17. Let P and Q be *hnfs*. We write $P \prec Q$ if $\mathcal{IA}(Q) \subseteq \mathcal{IA}(P)$ and one of the following conditions holds:

1. $P \equiv \sum_{e \in A} g_1(e)$ and $Q \equiv \sum_{e \in A} g_2(e)$;
2. $P \equiv \sum_{A \in \mathcal{A}} \sum_{e \in A} g_1(e)$ and $Q \equiv \sum_{B \in \mathcal{B}} \sum_{e \in B} g_2(e)$, with $\mathcal{B} \subset \mathcal{A}$;
3. $P \equiv \sum_{A \in \mathcal{A}} \sum_{e \in A} g_1(e)$ and $Q \equiv \sum_{e \in B} g_2(e)$, with $A \triangleleft B$ for some $A \in \mathcal{A}$.

Proposition 6.18. If P and Q are *hnfs* then $P \sqsubseteq_M Q$ implies $P \prec Q$.

Proof. We start proving that $\mathcal{IA}(Q) \subseteq \mathcal{IA}(P)$. We show that if there exists $\alpha \in \mathcal{IA}(Q) \setminus \mathcal{IA}(P)$ then $P \not\sqsubseteq_M Q$. Suppose that $\alpha = ot! \in \mathcal{IA}(Q) \setminus \mathcal{IA}(P)$ and take the observer $O \equiv \mathbf{in}(ot).\mathbf{nil}[] \text{isucc}$. Since $P \downarrow \varepsilon$ (indeed P is a *hnf*) and $\alpha \notin \mathcal{IA}(P)$, i.e. $\alpha \notin L(P)$, then P must O . However, we can construct the unsuccessful computation $Q|O \rightarrow^* Q_{ot!}|\mathbf{nil}$ hence Q must O . If $\alpha = ot?$ we can argue similarly by using the observer $(\mathbf{out}(ot).\mathbf{nil})[] \text{isucc}$. Therefore, $\mathcal{IA}(Q) \subseteq \mathcal{IA}(P)$ and we are left to prove that one of the conditions in Definition 6.17 holds.

Suppose that $P \equiv \sum_{e \in A} g_1(e)$. The hypothesis $P \ll_M Q$ implies $\mathcal{A}(Q, \varepsilon) \subset \mathcal{A}(P, \varepsilon) = \{A\}$, i.e. $\forall B \in \mathcal{A}(Q, \varepsilon): A \triangleleft B$. This implies $A \triangleleft Ev(\mathcal{B})$ and then $A \triangleleft cl(Ev(\mathcal{B}))$. Since $\mathcal{IA}(Q) \subseteq \mathcal{IA}(P)$ then $cl(Ev(\mathcal{B})) \triangleleft cl(A) = A$. Hence, $A = cl(Ev(\mathcal{B}))$ and since \mathcal{B} is saturated $A \in \mathcal{B}$. Therefore, for each $B \in \mathcal{B}$, $A = cl(Ev(\mathcal{B})) \triangleleft B \triangleleft cl(Ev(\mathcal{B})) = A$, i.e. $\mathcal{B} = \{A\}$, and Q must have the form $\sum_{e \in A} g_2(e)$ and condition 1 of Definition 6.17 holds. Suppose now that $P \equiv \sum_{A \in \mathcal{A}} \sum_{e \in A} g_1(e)$. We have two cases to consider according to the syntactic form of Q . First, assume that $Q \equiv \sum_{B \in \mathcal{B}} \sum_{e \in B} g_2(e)$. We must prove that condition 2 of Definition 6.17 holds, i.e. that $\mathcal{B} \subset \mathcal{A}$. This directly follows from the fact that \sqsubseteq_M and \ll_M coincide. Indeed, by definition of \ll_M , the hypothesis $P \sqsubseteq_M Q$ implies that $\mathcal{B} = \mathcal{A}(Q, \varepsilon) \subset \mathcal{A}(P, \varepsilon) = \mathcal{A}$. Suppose now that $Q \equiv \sum_{e \in B} g_2(e)$. We must prove that condition 3 of Definition 6.17 holds, i.e. $\exists A \in \mathcal{A}: A \triangleleft B$. From the syntactic form of Q it follows that $\mathcal{A}(Q, \varepsilon) = \{B\}$. Since $P \ll_M Q$, then $A \in \mathcal{A}$ exists such that $A \triangleleft B$. \square

Proposition 6.19. Let P and Q be *hnfs* such that $P \prec Q$; then $P \sqsubseteq_{\mathcal{B}\mathcal{P}} Q$ if for each $\alpha \in \mathcal{IA}(Q)$, $P_\alpha \sqsubseteq_{\mathcal{B}\mathcal{P}} Q_\alpha$.

Proof. Since $\mathcal{IA}(Q) \subseteq \mathcal{IA}(P)$ then $S(Q) \triangleleft S(P)$. Let g_1 and g_2 be the normal functions associated with P and Q , respectively. Let us consider the process R defined by substituting in Q each $g_2(e)$ with $g_3(e)$ where $g_3: S(Q) \rightarrow \mathcal{P}roc$, by using Notation 6.1, is defined as follows:

- if $e = (o, ot)$ then $g_3(e) = g_1(e)$;
- if $e = (i, p)$ then $g_3(e) = \mathbf{in}(it).E\sigma(it, it')$ where it , it' and E are such that $g_2(e) = \mathbf{in}(it).F$ for some F , $it' \Leftarrow it$, $\exists e' \in S(P): \{e\} \triangleleft \{e'\}$ and, by possibly applying rule IX, $g_1(e') = \mathbf{in}(it').E$.

By using the hypothesis and by applying rule II (case \parallel) for α of the form $ot!$ and rule III for α of the form $ot?$, it is straightforward to show that $g_3(e) \sqsubseteq_{\mathcal{B}\mathcal{P}} g_2(e)$ for

each $e \in S(Q)$. Therefore, by using rule II, we deduce that $R \sqsubseteq_{\mathcal{RP}} Q$. If P is of the form $\sum_{e \in A} g_1(e)$ then $P \equiv R$ and we have finished otherwise we use laws D3 and D4 (in the reverse direction of the saturation procedure described in Proposition 6.15) for rewriting P in the form $\sum_{A \in (\mathcal{A} \setminus \mathcal{B})} \sum_{e \in A} g_1(e) \oplus \sum_{B \in \mathcal{B}} \sum_{e \in B} g_3(e)$ where \mathcal{A} and \mathcal{B} are the saturated sets associated with P and Q , respectively. By applying law IC4 it immediately follows that $P \sqsubseteq_{\mathcal{RP}} Q$ and the thesis is proven. \square

We can now prove partial completeness of \mathcal{RP} .

Theorem 6.20. *For all processes P and Q , P finite, $P \sqsim_M Q$ implies $P \sqsubseteq_{\mathcal{RP}} Q$.*

Proof. If $P \uparrow \varepsilon$ then $P =_{\mathcal{RP}} \Omega$ (Proposition 6.12) and the thesis follows from law UND1. Otherwise, $P \downarrow \varepsilon$ and therefore there exists a *hnf*, $h(P)$ such that $P =_{\mathcal{RP}} h(P)$ (Proposition 6.15). Soundness of \mathcal{RP} with respect to \ll_M implies that P and $h(P)$ have exactly the same traces and, then, $\text{depth}(P) = \text{depth}(h(P))$. The hypothesis implies that $Q \downarrow \varepsilon$, hence, by Proposition 6.15, there exists a *hnf*, $h(Q)$ such that $Q =_{\mathcal{RP}} h(Q)$. We are left to prove that $h(P) \sqsubseteq_{\mathcal{RP}} h(Q)$. We proceed by induction on $\text{depth}(P)$. If $\text{depth}(P) = 0$ then $h(P) \equiv \text{nil}$. Because of soundness of \mathcal{RP} , the hypothesis implies $h(P) \ll_M h(Q)$. Therefore it must be $h(Q) \equiv \text{nil}$ and the thesis for this part is proven. Suppose now that $\text{depth}(h(P)) > 0$. From Proposition 6.18, we have that $h(P) \prec h(Q)$. In particular, this means that $\mathcal{IA}(h(Q)) \subseteq \mathcal{IA}(h(P))$, hence $\mathcal{IA}(h(Q)) \cap \mathcal{IA}(h(P)) = \mathcal{IA}(h(Q))$. By Proposition 6.16, we deduce that for each $\alpha \in \mathcal{IA}(h(Q))$: $h(P)_\alpha \sqsim_M h(Q)_\alpha$. By the inductive hypothesis, we deduce that for each $\alpha \in \mathcal{IA}(h(Q))$: $h(P)_\alpha \sqsubseteq_{\mathcal{RP}} h(Q)_\alpha$. From Proposition 6.19, we obtain the thesis. \square

Now we consider the full proof system \mathcal{CP} . The proposition below (that relies on partial completeness of \mathcal{RP}) will be used for proving soundness of \mathcal{CP} . There, P^n denotes the n th finite syntactic approximant of P .

Proposition 6.21. *For any process P and any observer O , if P must O then there exists $n \geq 0$ such that P^n must O .*

Proof. To prove the thesis it suffices to show that there exists a finite R such that $R \sqsubseteq_{\mathcal{RP}} P$ and R must O (it is, indeed, routine (see, e.g., [28]) to show that, for any finite process P and any process Q , $P \sqsubseteq_{\mathcal{RP}} Q$ implies $P \sqsubseteq_{\mathcal{RP}} Q^n$ for some $n \geq 0$). Suppose now that P must O and consider the computation tree from $P|O$ with branches pruned to obtain a tree whose leaves are all those nodes that can perform ω ; call this tree T . Since, for each process R , $ID(R)$ is finite (Proposition 4.4), then T is finitely branching. Hence, since P must O , by König's lemma, it follows that T is finite. The proof now proceeds by induction on the maximal number of communications between P and O in a path from the root to a leaf in T . It can be easily seen that this number does not change if we consider the computation tree from $P'|O$ for any P' that is testing equivalent to P . If $P \uparrow \varepsilon$ then it must be that $O \xrightarrow{\omega}$ and we can take $R = \Omega$. If $P \downarrow \varepsilon$

then (using partial completeness of \mathcal{RP}) we may suppose that P is in *hnf* and we can reason by case analysis.

Suppose that $P \equiv \sum_{e \in A} g(e)$. A process R is built up out of the finite processes R_e , for $e \in A$, defined as follows.

$e = (i, p)$. Let P_e and it be such that $g(e) = \mathbf{in}(it).P_e$. Consider the set

$$\text{Pairs}(it) = \{(ot, O') \mid O \rightarrow O_1 \rightarrow \dots \rightarrow O_k \xrightarrow{ot!} O', \text{match}(it, ot), \forall 0 \leq j \leq k: O_j \not\rightarrow\}$$

For each $(ot, O') \in \text{Pairs}(it)$, we have $P|O \rightarrow^* P_e[ot/it]|O'$, thus the hypothesis implies that $P_e[ot/it]$ must $\sum \{O'' \mid (ot, O'') \in \text{Pairs}(it)\}$. By the inductive hypothesis, there exists a finite process $R_{e,ot}$ such that $R_{e,ot} \sqsubseteq_{\mathcal{RP}} P_e[ot/it]$ and $R_{e,ot}$ must $\sum \{O'' \mid (ot, O'') \in \text{Pairs}(it)\}$, i.e. $R_{e,ot}$ must O'' for each $(ot, O'') \in \text{Pairs}(it)$. Let $mt(e) = \{ot \mid \exists O' : (ot, O') \in \text{Pairs}(it)\}$. Since, for each R , $OT(R)$ is finite (Proposition 4.4), then $mt(e)$ is finite, say $mt(e) = \{ot_1, ot_2, \dots, ot_n\}$. We can define

$$R_e = \mathbf{if} \text{ bem}(it, ot^1) \text{ then } R_{e,ot^1} \text{ else } \dots \mathbf{if} \text{ bem}(it, ot^n) \text{ then } R_{e,ot^n} \text{ else } \Omega$$

By rule VII it is easy to check that $R_e[ot/it] \sqsubseteq_{\mathcal{RP}} P_e[ot/it]$, $\forall ot : \text{match}(it, ot)$.

$e = (o, ot)$ Let P_e be such that $g(e) = \mathbf{out}(ot)\mathbf{nil} \parallel P_e$. If the set

$$\text{Get}(ot) = \{O' \mid O \rightarrow O_1 \rightarrow \dots \rightarrow O_k \xrightarrow{ot?} O', \forall 0 \leq j \leq k: O_j \not\rightarrow\}$$

is not empty, then P_e must $\sum \{O' \mid O' \in \text{Get}(ot)\}$. By the inductive hypothesis, there exists a finite process R'_e such that $R'_e \sqsubseteq_{\mathcal{RP}} P_e$ and R'_e must $\sum \{O' \mid O' \in \text{Get}(ot)\}$, i.e. R'_e must O' for each $O' \in \text{Get}(ot)$. In this case we define $R_e \equiv R'_e$.

If $\text{Get}(ot) = \emptyset$ then we define $R_e \equiv \Omega$.

Now we define $R \equiv \sum_{e \in A} g'(e)$ where $\forall e \in A : g'(e) \equiv g(e)[R_e/P_e]$. By construction, R must O and, moreover, by using rules II and III, it can be easily seen that $R \sqsubseteq_{\mathcal{RP}} P$.

Suppose that $\sum_{A \in \mathcal{A}} \sum_{e \in A} g(e)$. For each $A \in \mathcal{A}$, $\sum_{e \in A} g(e)$ must O . By repeating the above construction, we get that, for each $A \in \mathcal{A}$, there exists a finite process R_A such that $R_A \sqsubseteq_{\mathcal{RP}} \sum_{e \in A} g(e)$ and R_A must O . We define $R \equiv \sum_{A \in \mathcal{A}} R_A$ and the thesis follows (by using rule II). \square

Theorem 6.22. For all processes P and Q , $P \sqsubseteq_{\mathcal{RP}} Q$ implies $P \sqsubseteq_{\mathcal{M}} Q$.

Proof. We are only left to show that rules IV(a) and VI are sound. Rule IV(a) is derivable from the other axioms and rules in \mathcal{CP} (see, e.g., [19]) and therefore it is sound. Soundness of the ω -inductive rule VI easily follows by Proposition 6.21. \square

Finally, completeness of the full proof system can be established.

Theorem 6.23. For all processes P and Q , $P \sqsubseteq_{\mathcal{M}} Q$ implies $P \sqsubseteq_{\mathcal{RP}} Q$.

Proof. Suppose that $P \sqsubseteq_{\mathcal{M}} Q$. A standard result of algebraic semantics (see, e.g., [28]) states that since $\sqsubseteq_{\mathcal{M}}$ satisfies I, II, III, IV(b) and UND1 then for every $n \geq 0 : P^n \sqsubseteq_{\mathcal{M}} Q$.

By completeness of \mathcal{RP} , we can infer that for every $n \geq 0$, $P^n \sqsubseteq_{\mathcal{RP}} Q$ and, then, for every $n \geq 0$, $P^n \sqsubseteq_{\mathcal{CP}} Q$. Now, we can apply VI and conclude that $P \sqsubseteq_{\mathcal{CP}} Q$. \square

6.1. Using the proof system

In this section, we show how the proof system can be used for proving correctness of simple programs that permit adding two arrays elementwise. We already assumed that the language is parameterized on a countable set of values, *Val*. Here, we require that *Val* be the *Natural Numbers*. Let A and B be two arrays of n naturals. We shall consider PAL processes which add A and B elementwise leaving the result in an array C .

To better exploit the parallelism intrinsic in the problem, every array is represented as a *distributed data structure* [13]. Hence, we shall use a tuple for every single element of every array. To represent A, B, C we shall use n tuples with three fields: the first one contains a constant 0, 1, 2 which identifies the array, the second one the index i of the element and the last one the value A_i, B_i, C_i of the element.

The processes we consider add elements of A and B with the same index, if they both exist, for any (finite) length arrays. Let us consider the PAL processes

- $Q \equiv \text{rec}X.\text{in}(0, \underline{x}, \underline{y}).\text{eval}(X).\text{in}(1, \underline{x}, \underline{z}).\text{out}(2, \underline{x}, \underline{y} + \underline{z}).\text{nil}$, and
- P_k where $P_2 \equiv Q|Q$ and $P_{j+1} \equiv Q|P_j$, for $j > 2$.

We want to show that process P_k obtained by putting in parallel k copies of Q is provably equal to Q , that is $P_k =_{\mathcal{CP}} Q$. We may think of Q as a process which executes on a single processor and is able to dynamically reproduce itself when a new element of array A is accessed. Each instance computes an element of array C . The number of instances which are concurrently active depends on the difference between the number of elements of A and that of elements of B which have been accessed. We may think about P_k as a really distributed process consisting of k copies of process Q which are simultaneously executed on k different processors. In this sense P_k may be thought of as a more efficient and fault-tolerant solution of the problem than Q .

Rather than using the full power of \mathcal{CP} , we will use a simpler induction rule. Here we use a powerful but simple form of induction for dealing with recursively defined terms, namely *Unique Fixpoint Induction* [28], which is expressed by the following rule:

$$\text{UFI} \quad \frac{E = F[E/X]}{E = \text{rec}X. F} \quad \text{where } X \text{ is guarded.}$$

UFI can be derived within \mathcal{CP} (see, e.g. [19]) and here it can be correctly used since all terms we examine are *guarded*, i.e. all occurrences of process variables are preceded by a blocking prefix.

We proceed by induction on k . Firstly, we prove that $P_2 =_{\mathcal{CP}} Q$. The term Q is recursively defined and in order to show

$$P_2 = \text{rec}X.\text{in}(0, \underline{x}, \underline{y}).\text{eval}(X).\text{in}(1, \underline{x}, \underline{z}).\text{out}(2, \underline{x}, \underline{y} + \underline{z}).\text{nil}$$

we can use an instance of UFI. Hence, it is sufficient to deduce

$$P_2 = \mathbf{in}(0, \underline{x}, \underline{y}).\mathbf{eval}(P_2).\mathbf{in}(1, x, \underline{z}).\mathbf{out}(2, x, y + z).\mathbf{nil}$$

that is, by applying EVAL,

$$P_2 = \mathbf{in}(0, \underline{x}, \underline{y}).(P_2|\mathbf{in}(1, x, \underline{z}).\mathbf{out}(2, x, y + z).\mathbf{nil}). \quad (1)$$

This may be proven by using a standard strategy, consisting of expanding out the recursive definitions and applying the laws for parallel operators and the interleaving law PAR3.

By expanding out the recursive definitions and applying EVAL we get

$$\begin{aligned} P_2 &\equiv Q|Q \\ &= \mathbf{in}(0, \underline{x}_1, \underline{y}_1).(Q|\mathbf{in}(1, x_1, \underline{z}_1).\mathbf{out}(2, x_1, y_1 + z_1).\mathbf{nil}) \\ &\quad |\mathbf{in}(0, \underline{x}_2, \underline{y}_2).(Q|\mathbf{in}(1, x_2, \underline{z}_2).\mathbf{out}(2, x_2, y_2 + z_2).\mathbf{nil}). \end{aligned}$$

Since

$$Q = \mathbf{in}(0, \underline{x}_1, \underline{y}_1).(Q|\mathbf{in}(1, x_1, \underline{z}_1).\mathbf{out}(2, x_1, y_1 + z_1).\mathbf{nil})$$

and

$$Q = \mathbf{in}(0, \underline{x}_2, \underline{y}_2).(Q|\mathbf{in}(1, x_2, \underline{z}_2).\mathbf{out}(2, x_2, y_2 + z_2).\mathbf{nil}),$$

by applying the interleaving law PAR3 we get

$$\begin{aligned} P_2 &= \mathbf{in}(0, \underline{x}_1, \underline{y}_1).(Q|Q|\mathbf{in}(1, x_1, \underline{z}_1).\mathbf{out}(2, x_1, y_1 + z_1).\mathbf{nil}) \\ &\quad []\mathbf{in}(0, \underline{x}_2, \underline{y}_2).(Q|Q|\mathbf{in}(1, x_2, \underline{z}_2).\mathbf{out}(2, x_2, y_2 + z_2).\mathbf{nil}). \end{aligned}$$

By applying IX for renaming the variables bound by input prefixes and EC3 for coalescing the two summands of $[]$ we get Eq. (1) and then we conclude that $P_2 =_{\mathcal{E}} Q$.

Now, we prove the inductive step. We assume that $P_j =_{\mathcal{E}} Q$. By applying II with hypothesis $Q \sqsubseteq Q$ and $P_j \sqsubseteq Q$ in the case of the operator $|$ we get

$$P_{j+1} \equiv Q|P_j \sqsubseteq Q|Q \equiv P_2 = Q.$$

In a similar way we can derive that $Q \sqsubseteq P_{j+1}$ and therefore we conclude that $P_k =_{\mathcal{E}} Q$ for all $k \geq 2$.

7. Denotational semantics for PAL

In this section we define a denotational semantics for PAL and prove that it is *fully abstract* with respect to the testing preorders. The denotational model shall be given under the form of a *natural interpretation*. This is a slight variant of the usual *algebraic semantics* (see, e.g. [25,28]) and it has been introduced in [31] for dealing

with languages with value-passing (see, e.g. [31,29,35]). Much of the following notation is borrowed from [29,35].

Notation 7.1. We will write $\lambda x.[be(x)] \rightarrow g(x), h(x)$ to denote a function f which is defined by $f(x) = g(x)$ if $be(x)$, $f(x) = h(x)$ otherwise.

Let A_1, A_2, B_1 and B_2 be sets and $f_j: A_j \rightarrow B_j$, $j = 1, 2$. Let \uplus denote disjoint union between sets. We let $f_1 \uplus f_2$ denote the function with functionality $A_1 \uplus A_2 \rightarrow B_1 \uplus B_2$ defined by

$$(f_1 \uplus f_2)(a) = \begin{cases} f_1(a) & \text{if } a \in A_1, \\ f_2(a) & \text{if } a \in A_2. \end{cases}$$

Let f be a function from a cpo (complete partial order) D to a cpo D' ; we will use $dom(f)$ to refer to D (the domain of f) and we let $support(f) = \{d \in dom(f) \mid f(d) \neq \perp\}$. θ will be used to denote the empty (totally undefined) function (i.e., if θ is a function from D to D' , $dom(\theta) = D$ and $support(\theta) = \emptyset$). If I is a set of elements of a cpo $\langle D, \leqslant_D \rangle$, we will use $\sqcup I$ for denoting the least upper bound (lub) of I with respect to \leqslant_D , if it exists.

We let $Fin(S \mapsto D)$ denote the set of partial functions from a set S to a cpo $\langle D, \leqslant_D \rangle$ with finite domain, and let $Fin_s(S \mapsto D)$ denote those functions with finite support. On both the sets, we will use the following non-standard ordering:

$$f \trianglelefteq_D g \text{ iff } dom(g) \subseteq dom(f) \text{ and } \forall s \in support(f) \cap dom(g): f(s) \leqslant_D g(s).$$

Note that the more defined the partial function is the smaller it is for \trianglelefteq_D . It holds that if $\langle D, \leqslant_D \rangle$ is a (ω -algebraic) cpo then $\langle Fin(S \mapsto D)_\perp, \trianglelefteq_D \rangle$ and $\langle Fin_s(S \mapsto D)_\perp, \trianglelefteq_D \rangle$ are (ω -algebraic) cpo's as well (the proof can be done along the lines of that of Lemma 3.3.5 in [35]).

7.1. The model AT^L : acceptance trees for Linda

Due to its computational nature, we choose to interpret the language in some cpo D where recursive definitions can be interpreted. To each of the operators in Σ^{-ir} we associate a continuous function over D of appropriate arity. The only exceptions are the **eval** prefixing, which is a derived operator and whose denotational semantics is given by using the parallel operator, and the **if be then _ else _**. For the sake of simplicity, in the following we will use Σ^- to denote the set of all PAL operators with the exception of prefixing by **in, read** and **eval**, and of the conditional construct **if be then _ else _**.

The input prefixes cannot be interpreted similarly, as they are binding operators for value variables, and we need an extra structure for interpreting them. For example, when $var(t) \neq \emptyset$, **in**(t)._ can take an open term and return a process. An appropriate type for an input operation is

$$AEIT \times (EOT \rightarrow D) \rightarrow D$$

where D is the proposed interpretation of processes. $\mathbf{read}(t)_{\cdot}$ can be interpreted similarly.

Definition 7.2. A natural interpretation for PAL is a quadruple $\langle D, \leqslant_D, \Sigma_D^-, in_D \rangle$ where

- $\langle D, \leqslant_D, \Sigma_D^- \rangle$ is a Σ -cpo, i.e. $\langle D, \leqslant_D \rangle$ is a cpo on which is defined a continuous function for each operator in the signature Σ^- ,
- $in_D: AEIT \times (EOT \rightarrow D) \rightarrow D$ is a total function continuous in its second argument, (where $(EOT \rightarrow D)$ inherits the natural pointwise ordering, denoted by \sqsubseteq_D , from D).

Given such a natural interpretation D , we can define a denotational semantics for PAL. To cope with open terms, we use D -environments, i.e. mappings from \mathcal{X} (the set of process variables) to D . Env_D , ranged over by ξ , will represent the set of D -environments.

The denotational semantics is given as a function $\mathcal{D}[\cdot]: \text{PAL} \rightarrow (Env_D \rightarrow D)$ defined by structural induction via the following clauses:

1. $\mathcal{D}[X]\xi = \xi(X)$
2. $\mathcal{D}[\Omega]\xi = \Omega_D$
3. $\mathcal{D}[\mathbf{nil}]\xi = nil_D$
4. $\mathcal{D}[Eop F]\xi = \mathcal{D}[E]\xi op_D \mathcal{D}[F]\xi$ where $op \in \{ \oplus, [], |, \lfloor, \rfloor \}$
5. $\mathcal{D}[\mathbf{eval}(E).F]\xi = \mathcal{D}[E]\xi|_D \mathcal{D}[F]\xi$
6. $\mathcal{D}[\mathbf{out}(t).E]\xi = out_D^{ot}(\mathcal{D}[E]\xi)$ where $ot = \mathcal{O}[t]$
7. $\mathcal{D}[\mathbf{if} \text{ be then } E \text{ else } F]\xi = \mathcal{D}[E]\xi$ if $\mathcal{B}[be] = tt$
 $\mathcal{D}[\mathbf{if} \text{ be then } E \text{ else } F]\xi = \mathcal{D}[F]\xi$ if $\mathcal{B}[be] = ff$
8. $\mathcal{D}[\mathbf{rec} X.E]\xi = Y\lambda d. \mathcal{D}[E]\xi[d/X]$
9. $\mathcal{D}[\mathbf{in}(t).E]\xi = in_D(\wp(\mathcal{J}[t]), g)$ where $g = \lambda x. [match(\mathcal{J}[t], x)] \rightarrow \mathcal{D}[E[x/\mathcal{J}[t]]]\xi, \perp_D$
 $\mathcal{D}[\mathbf{read}(t).E]\xi = in_D(\wp(\mathcal{J}[t]), g)$ where $g = \lambda x. [match(\mathcal{J}[t], x)] \rightarrow (out_D^x(nil_D))|_D$
 $\mathcal{D}[E[x/\mathcal{J}[t]]]\xi, \perp_D$

where Y is the least fixed point operator for continuous functions in D .

In the rest of this section we shall construct a particular natural interpretation AT^L that properly reflects the testing preorder \sqsubseteq_M . It rests on a ω -algebraic Σ -cpo, i.e. an algebraic cpo with a countable set of compact elements. As the interpretation is algebraic, it is completely determined by its compact elements.

The construction of the model AT^L rests on the description of the set fAT^L of its compact elements and on the description of the relative partial ordering \leqslant_{fAT^L} .

Definition 7.3 (Compact elements). We define the cpo $\langle fAT^L, \leqslant_{fAT^L} \rangle$ by

- fAT^L is the least set that satisfies the following requirements:
 1. $\perp \in fAT^L$
 2. if $\mathcal{A} \in sat(Ev)$, $f_{out} \in Fin(EOT \mapsto fAT^L)_{\perp}$, $dom(f_{out}) = \mathcal{T}(Ev_{out}(\mathcal{A}))$, $f_{ir} \in Fin_s(EOT \mapsto fAT^L)_{\perp}$ and $dom(f_{ir}) = \mathcal{MT}(\mathcal{P}(Ev_{ir}(\mathcal{A})))$ then $\langle \mathcal{A}, f_{ir} \uplus f_{out} \rangle \in fAT^L$.

• \leq_{fAT^L} is defined as follows:

1. $\perp \leq_{fAT^L} T$ for all $T \in fAT^L$
2. $\langle \mathcal{A}, f_{ir} \uplus f_{out} \rangle \leq_{fAT^L} \langle \mathcal{B}, g_{ir} \uplus g_{out} \rangle$ if $\mathcal{B} \subset \mathcal{A}$, $f_{ir} \trianglelefteq_{fAT^L} g_{ir}$ and $f_{out} \trianglelefteq_{fAT^L} g_{out}$.

We shall write $f_{ir} \uplus f_{out} \leq_{fAT^L} g_{ir} \uplus g_{out}$ as shorthand for $f_{ir} \trianglelefteq_{fAT^L} g_{ir}$ and $f_{out} \trianglelefteq_{fAT^L} g_{out}$. Note that the less deterministic the process is the less is it in the order.

Now, we turn the poset fAT^L in a Σ -po algebra by providing a Σ -algebra structure to it. To this aim we must define a monotonic function for each operator in Σ^- and, in addition, an input function of the correct type monotonic in its second argument.

We start defining the special function in_{fAT^L} for input prefixes; then, we shall define the functions in $\Sigma_{fAT^L}^-$.

in_{fAT^L} : Define $in_{fAT^L} : AEIT \times Fin_s(EOT \rightarrow fAT^L) \rightarrow fAT^L$ by

$$in_{fAT^L}(p, g) = \langle \{ \{ (i, p) \} \}, f \uplus \theta \rangle$$

where $dom(f) = \mathcal{MT}(\{p\})$ and $\forall ot \in dom(f) : f(ot) = g(ot)$.

nil_{fAT^L} : Let nil_{fAT^L} be the tree $\langle \{\emptyset\}, \theta \uplus \theta \rangle$.

Ω_{fAT^L} : Let Ω_{fAT^L} be the tree \perp .

$out_{fAT^L}^{ot}$: For every $ot \in EOT$, define $out_{fAT^L}^{ot} : fAT^L \rightarrow fAT^L$ by

$$out_{fAT^L}^{ot}(T) = \langle \{ \{ (o, ot) \} \}, \theta \uplus f \rangle$$

where $dom(f) = \{ot\}$ and $f(ot) = T$.

\oplus_{fAT^L} (internal choice): Define $\oplus_{fAT^L} : (fAT^L \times fAT^L) \rightarrow fAT^L$ as

$\lambda T. \lambda U.$ if $T = \perp$ or $U = \perp$ then \perp

else let $T = \langle \mathcal{A}, f_{ir} \uplus f_{out} \rangle$ and $U = \langle \mathcal{B}, g_{ir} \uplus g_{out} \rangle$

in $\langle sat(\mathcal{A} \cup \mathcal{B}), h_{ir} \uplus h_{out} \rangle$

where $\forall ot \in \mathcal{T}(Ev_{out}(sat(\mathcal{A} \cup \mathcal{B})))$, if $\gamma = out$, and $\forall ot \in \mathcal{MT}(\mathcal{P}(Ev_{ir}(sat(\mathcal{A} \cup \mathcal{B}))))$, if $\gamma = ir$, it holds that

$$h_{\gamma}(ot) = \begin{cases} f_{\gamma}(ot) \oplus_{fAT^L} g_{\gamma}(ot) & \text{if } ot \in dom(f_{\gamma}) \cap dom(g_{\gamma}) \\ f_{\gamma}(ot) & \text{if } ot \in dom(f_{\gamma}) \setminus dom(g_{\gamma}) \\ g_{\gamma}(ot) & \text{if } ot \in dom(g_{\gamma}) \setminus dom(f_{\gamma}) \end{cases}$$

\sqcap_{fAT^L} (external choice): Define $\sqcap_{fAT^L} : (fAT^L \times fAT^L) \rightarrow fAT^L$ as

$\lambda T. \lambda U.$ if $T = \perp$ or $U = \perp$ then \perp

else let $T = \langle \mathcal{A}, f_{ir} \uplus f_{out} \rangle$ and $U = \langle \mathcal{B}, g_{ir} \uplus g_{out} \rangle$

in $\langle sat(\mathcal{A} \vee \mathcal{B}), h_{ir} \uplus h_{out} \rangle$

where $\mathcal{A} \vee \mathcal{B}$ is the pointwise union of \mathcal{A} and \mathcal{B} , i.e. the set $\{A \cup B \mid A \in \mathcal{A}, B \in \mathcal{B}\}$, and h_{ir} and h_{out} are defined as in the case of the \oplus_{fAT^L} operator.

$|_{fAT^L}$ (parallel composition): Define $|_{fAT^L} : (fAT^L \times fAT^L) \rightarrow fAT^L$ as

$$\begin{aligned} \lambda T. \lambda U. \text{ if } T = \perp \text{ or } U = \perp \text{ then } \perp \\ \text{ else let } T = \langle \mathcal{A}, f_{ir} \uplus f_{out} \rangle \text{ and } U = \langle \mathcal{B}, g_{ir} \uplus g_{out} \rangle \\ \text{ in } \sum_{fAT^L} \{ T_{AB} \mid A \in \mathcal{A}, B \in \mathcal{B} \} \end{aligned}$$

where

$$\begin{aligned} T_{AB} &= \text{if } INT(A, B) = \emptyset \\ &\text{ then } \text{sumext}(A, B) \\ &\text{ else } (\text{sumext}(A, B) []_{fAT^L} \text{sumint}(A, B)) \oplus_{fAT^L} \text{sumint}(A, B) \\ \text{sumext}(A, B) &= \sum_{fAT^L} EXT(A, B) \\ \text{sumint}(A, B) &= \sum_{fat^L} INT(A, B) \end{aligned}$$

$$\begin{aligned} INT(A, B) &= \{ f_{ir}(ot) |_{fAT^L} g_{out}(ot) \mid ot \in (\mathcal{T}(Ev_{out}(B)) \cap \mathcal{MT}(\mathcal{P}(Ev_{ir}(A)))) \} \\ &\cup \{ f_{out}(ot) |_{fAT^L} g_{ir}(ot) \mid ot \in (\mathcal{T}(Ev_{out}(A)) \cap \mathcal{MT}(\mathcal{P}(Ev_{ir}(B)))) \} \end{aligned}$$

$$\begin{aligned} EXT(A, B) &= \{ in_{fAT^L}(p, f') \mid p \in \mathcal{P}(Ev_{ir}(A)), \text{dom}(f') = \text{dom}(f_{ir}), \\ &\quad \forall ot \in \text{dom}(f') : f'(ot) = f_{ir}(ot) |_{fAT^L} U \} \\ &\cup \{ in_{fat^L}(p, g') \mid p \in \mathcal{P}(Ev_{ir}(B)), \text{dom}(g') = \text{dom}(g_{ir}), \\ &\quad \forall ot \in \text{dom}(g') : g'(ot) = T |_{fat^L} g_{ir}(ot) \} \\ &\cup \{ out_{fAT^L}^{ot}(f_{out}(ot) |_{fAT^L} U) \mid ot \in \mathcal{T}(Ev_{out}(A)) \} \\ &\cup \{ out_{fat^L}^{ot}(T |_{fat^L} g_{out}(ot)) \mid ot \in \mathcal{T}(Ev_{out}(B)) \}. \end{aligned}$$

$\|_{fAT^L}$ (communication-merge): Define $\|_{fAT^L} : (fAT^L \times fAT^L) \rightarrow fAT^L$ as

$$\begin{aligned} \lambda T. \lambda U. \text{ if } T = \perp \text{ or } U = \perp \text{ then } \perp \\ \text{ else let } T = \langle \mathcal{A}, f_{ir} \uplus f_{out} \rangle \text{ and } U = \langle \mathcal{B}, g_{ir} \uplus g_{out} \rangle \\ \text{ in } \sum_{fAT^L} \{ \text{sumint}(A, B) \mid A \in \mathcal{A}, B \in \mathcal{B} \} \end{aligned}$$

where $\text{sumint}(A, B)$ is defined as in the case of the $|_{fAT^L}$ operator.

\llcorner_{fAT^L} (left-merge): Define $\llcorner_{fAT^L} : (fAT^L \times fAT^L) \rightarrow fAT^L$ as

$$\begin{aligned} \lambda T. \lambda U. \text{ if } T = \perp \text{ then } \perp \\ \text{ else let } T = \langle \mathcal{A}, f_{ir} \uplus f_{out} \rangle \\ \text{ in } \langle \mathcal{A}, h_{ir} \uplus h_{out} \rangle \end{aligned}$$

where $\text{dom}(h_{ir}) = \text{dom}(f_{ir})$ and $\text{dom}(h_{out}) = \text{dom}(f_{out})$, and $\forall ot \in \mathcal{T}(Ev_{out}(\mathcal{A}))$, if $\gamma = out$, and $\forall ot \in \mathcal{MT}(\mathcal{P}(Ev_{ir}(\mathcal{A})))$, if $\gamma = ir$, it holds that $h_{\gamma}(ot) = f_{\gamma}(ot) |_{fAT^L} U$.

Remark 7.4. Our ordering between partial functions and the distinction between domain and support of a (partial) function permit discriminating between functions “undefined in d ” and functions “evaluating to \perp in d ”. We can now show, by means of a simple example, why this discrimination is important for our theory. Consider the following two processes $P \equiv \mathbf{in}(7).\mathbf{nil}$ and $Q \equiv \mathbf{in}(z).\mathbf{if } z = 7 \text{ then nil else } \Omega$. It can be easily seen that $P \sqsubset_M Q$ and that $Q \sqsubset_M P$ (take the observers $\mathbf{out}(5).\mathbf{in}(5).\mathbf{success.nil}$ and $\mathbf{out}(5).\mathbf{nil} \parallel \mathbf{success.nil}$, respectively). However, if we adopt the standard denotational approach and look at “partial functions” as “global functions evaluating \perp outside their domain”, thus

$$\text{dom}(f) = \{ot \in EOT \mid f(ot) \neq \perp\}$$

and

$$f \sqsubseteq_{fAT^L} g \text{ iff } \text{dom}(g) \subseteq \text{dom}(f) \text{ and } \forall ot \in \text{dom}(g) : f(s) \leq_{fAT^L} g(s),$$

we would have $\mathcal{D}[P] = \langle \{(i, 7)\}, g_P \uplus \theta \rangle$ and $\mathcal{D}[Q] = \langle \{(i, -)\}, g_Q \uplus \theta \rangle$ where functions g_P and g_Q are *extensionally* equivalent to $g = \lambda x.[x = (7)] \rightarrow \mathbf{nil}_{fAT^L}, \perp$. Therefore, since $\{(i, -)\} \subset \{(i, 7)\}$, we could *wrongly* conclude that $\mathcal{D}[P] \leq_{fAT^L} \mathcal{D}[Q]$. In our approach, instead, we have $\text{dom}(g_Q) = \mathcal{MT}(\{(-)\}) = \{(v) \in EOT \mid v \in Val\} \not\subseteq \{(7)\} = \mathcal{MT}(\{(7)\}) = \text{dom}(g_P)$, hence $g_P \not\sqsubseteq_{fAT^L} g_Q$ and then $\mathcal{D}[P] \not\leq_{fAT^L} \mathcal{D}[Q]$.

Proposition 7.5. $\langle fAT^L, \leq_{fAT^L}, \Sigma_{fAT^L}^- \rangle$ is a Σ -predomain.

Proof. Since $\langle fAT^L, \leq_{fAT^L} \rangle$ is a poset with least element and since, by definition, Ω_{fAT^L} is the bottom element of fAT^L , we are only left to show that each function in $\Sigma_{fAT^L}^-$ is well-defined and monotonic. These proofs are routine. \square

Proposition 7.6. in_{fAT^L} is well-defined and monotonic in its second argument.

Proof. Recall that, by definition, if $p \in AEIT$ and $g \in \text{Fin}_s(EOT \rightarrow fAT^L)$ then $\text{in}_{fAT^L}(p, g) = \langle \{(i, p)\}, f \uplus \theta \rangle$ where $\text{dom}(f) = \mathcal{MT}(\{p\})$ and for all $ot \in \text{dom}(f)$: $f(ot) = g(ot)$. It is obvious that in_{fAT^L} is well-defined, i.e. $\text{in}_{fAT^L}(p, g) \in fAT^L$.

We must show that in_{fAT^L} is monotonic in its second argument. Let us assume that $g_1, g_2 \in \text{Fin}_s(EOT \rightarrow fAT^L)$ are such that $g_1 \leq_{fAT^L} g_2$ (recall that \leq_{fAT^L} is the pointwise ordering inherited from fAT^L). Define f_j , for $j = 1, 2$, by

- $\text{dom}(f_j) = \mathcal{MT}(\{p\})$,
- $f_j(ot) = g_j(ot)$, for all $ot \in \mathcal{MT}(\{p\})$.

Since $\text{dom}(f_1) = \text{dom}(f_2) = \mathcal{MT}(\{p\})$ and $\forall ot \in \text{support}(f_1) : f_1(ot) = g_1(ot) \leq_{fAT^L} g_2(ot) = f_2(ot)$, then $f_1 \sqsubseteq_{fAT^L} f_2$. The monotonicity of in_{fAT^L} in its second argument follows. \square

Now, we can use a standard technique of algebraic semantics, known as *completion by ideals*, for obtaining the algebraic cpo's $(fAT^L)^\infty$ and $(\text{Fin}_s(EOT \rightarrow fAT^L))^\infty$

and the unique continuous extensions of the previously defined monotonic functions [28, Theorem 3.3.10]. The crucial point is that the algebraic cpo's $\langle (Fin_s(EOT \rightarrow fAT^L))^\infty, \leq^{fAT^L} \rangle$ and $\langle (EOT \rightarrow (fAT^L)^\infty), \leq_{fAT^L} \rangle$ are isomorphic. Since, algebraic cpo's are completely determined by their compact elements, it suffices to show that $\langle Fin_s(EOT \rightarrow fAT^L), \leq_{fAT^L} \rangle$ and $\langle Comp(EOT \rightarrow (fAT^L)^\infty), \leq_{fAT^L} \rangle$ (where $Comp(EOT \rightarrow (fAT^L)^\infty)$ is the set of compact elements of $(EOT \rightarrow (fAT^L)^\infty)$) are isomorphic as partial orders; but this follows from the fact that fAT^L and $Comp((fAT^L)^\infty)$ are isomorphic.

The required process interpretation domain D is then $(fAT^L)^\infty$, that we will call AT^L ; this together with the extended functions give a natural interpretation of PAL processes. For the sake of simplicity we name the continuous extensions as the monotonic functions; they do extend, but replace the subscript fAT^L with AT^L .

The next two results follow directly from the standard algebraic semantics theory.

Corollary 7.7. $\langle AT^L, \leq_{AT^L}, \Sigma_{AT^L}^- \rangle$ is a Σ -domain.

Corollary 7.8. $\langle AT^L, \leq_{AT^L}, \Sigma_{AT^L}^-, in_{AT^L} \rangle$ is a natural interpretation.

7.2. Full abstraction of the denotational interpretation

In this section, we shall prove that the denotational model AT^L is fully abstract with respect to the proof-theoretic preorder $\sqsubseteq_{\mathcal{CP}}$, i.e. for all processes P and Q , $P \sqsubseteq_{\mathcal{CP}} Q$ if and only if $\mathcal{D}[P] \leq_{AT^L} \mathcal{D}[Q]$. Since we have already proven soundness and completeness of the proof system \mathcal{CP} with respect to the behavioural preorder \sqsubseteq_M , this will enable us to conclude that the denotational model AT^L is fully abstract with respect to the testing preorders.

We start by considering only finite processes and proving a full abstraction result for them; then we will generalize the result to general PAL processes.

The use of compact elements of the model will be essential. These elements, in general, do not correspond to finite processes because (finite) processes can input (one of) an infinite set of tuples, i.e. processes may not have finite breadth. Therefore, by relying on Notation 6.1, we introduce the notions of *finite-breadth approximants* and *compact* processes. We will show that the compact elements of the cpo are the semantic denotations of compact processes, and that every (recursively defined) process is semantically the limit of a directed set of compact processes.

Definition 7.9. For each finite process P , the set of *finite-breadth approximants* of P , $Fba(P)$ is inductively defined as follows

1. $Fba(\Omega) = \{\Omega\}$, $Fba(\mathbf{nil}) = \{\mathbf{nil}\}$,
2. $Fba(\mathbf{out}(t).Q) = \{\mathbf{out}(t).Q' \mid Q' \in Fba(Q)\}$,
3. $Fba(\mathbf{eval}(P_1).P_2) = \{\mathbf{eval}(Q_1).Q_2 \mid Q_1 \in Fba(P_1), Q_2 \in Fba(P_2)\}$,
4. $Fba(P_1 \text{ op } P_2) = \{Q_1 \text{ op } Q_2 \mid Q_1 \in Fba(P_1), Q_2 \in Fba(P_2)\}$, $\text{op} \in \{\oplus, \square, \mid, \parallel, \parallel\}$,
5. $Fba(\mathbf{if be then } P_1 \text{ else } P_2) = \{\mathbf{if be then } Q_1 \text{ else } Q_2 \mid Q_j \in Fba(P_j), j = 1, 2\}$,

6. $Fba(\mathbf{in}(t).E) = \{\mathbf{in}(t).Q(M) \mid it = \mathcal{I}[t], M \subseteq_{fin} \mathcal{MT}(\{\wp(it)\}), \forall ot^j \in M : Q_{ot^j} \in Fba(E[ot^j/it])\}$
 $Fba(\mathbf{read}(t).E) = \{\mathbf{read}(t).Q(M) \mid it = \mathcal{I}[t], M \subseteq_{fin} \mathcal{MT}(\{\wp(it)\}), \forall ot^j \in M : Q_{ot^j} \in Fba(E[ot^j/it])\}$

where $Q(M) \equiv \mathbf{if} \text{ bem}(it, ot^1) \text{ then } Q_{ot^1} \text{ else } \dots \mathbf{if} \text{ bem}(it, ot^n) \text{ then } Q_{ot^n} \text{ else } \Omega$
 whenever $M = \{ot^1, ot^2, \dots, ot^n\}$.

We shall say that a (finite) process P is *compact* when there exists a finite process Q such that $P \in Fba(Q)$.

By construction, we have that for any finite process P , if $Q \in Fba(P)$ then $Q \sqsubseteq_{\mathcal{AP}} P$.
 The following proposition states relevant properties used for proving full abstraction.

Proposition 7.10. *Let P and Q be finite processes; then*

1. $R \in Fba(P)$ implies $\mathcal{D}[R] \in fAT^L$, i.e. $\mathcal{D}[R]$ is compact in AT^L ;
2. $\{\mathcal{D}[R] \mid R \in Fba(P)\}$ is directed in AT^L and $\mathcal{D}[P] = \sqcup \{\mathcal{D}[R] \mid R \in Fba(P)\}$;
3. R is compact and $R \sqsubseteq_{\mathcal{AP}} P$ imply that there exists $R' \in Fba(P)$ such that $R \sqsubseteq_{\mathcal{AP}} R'$;
4. $\forall R \in Fba(P) : R \sqsubseteq_{\mathcal{AP}} Q$ implies $P \sqsubseteq_{\mathcal{AP}} Q$.

Proof (Outline)

1. It directly follows by structural induction from the definition of finite-breadth approximations of a process.
2. The proof proceeds by structural induction on P and exploits the continuity of the operators on AT^L . The only difficult cases are when $P \equiv \mathbf{read}(t).E$ and $P \equiv \mathbf{in}(t).E$ for those E such that $fv(E) \subseteq var(t)$. Here we only consider the former, the latter can be dealt with similarly.

By definition of Fba and of $\mathcal{D}[\cdot]$, and by structural induction

$$\begin{aligned} & \{\mathcal{D}[R] \mid R \in Fba(\mathbf{read}(t).E)\} \\ &= \{\mathcal{D}[\mathbf{read}(t).Q(M)] \mid M \subseteq_{fin} \mathcal{MT}(\{\wp(it)\}), it = \mathcal{I}[t], \\ & \quad \forall ot^j \in M : R_{ot^j} \in Fba(E[ot^j/it])\} \\ &= \{in_{AT^L}(\wp(it), g) \mid it = \mathcal{I}[t], g \in G\}, \end{aligned}$$

where $Q(M)$ is the process introduced in Definition 7.9(6) and

$$\begin{aligned} G &= \{g : EOT \rightarrow fAT^L \mid g \equiv (\lambda x.[x \in M] \rightarrow (out_{fAT^L}^x(nil_{fAT^L})|_{fAT^L} \mathcal{D}[R_x]), \perp), \\ & \quad M \subseteq_{fin} \mathcal{MT}(\{\wp(it)\}), \forall x \in M : R_x \in Fba(E[x/it])\}. \end{aligned}$$

Indeed, $T|_{fAT^L} \perp = \perp$ (by definition) and $M \subseteq_{fin} \mathcal{MT}(\{\wp(it)\})$ imply

$$\begin{aligned} & \lambda x.[match(it, x)] \rightarrow (out_{fAT^L}^x(nil_{fAT^L})|_{fAT^L} ([x \in M] \rightarrow \mathcal{D}[R_x]), \perp), \perp \\ &= \lambda x.[x \in M] \rightarrow (out_{fAT^L}^x(nil_{fAT^L})|_{fAT^L} \mathcal{D}[R_x]), \perp. \end{aligned}$$

By inductive hypothesis, for $x \in \mathcal{MT}(\{\wp(it)\})$, $\{\mathcal{D}[R_x] \mid R_x \in Fba(E[x/it])\}$, and then $\{(out_{fAT^L}^x(nil_{fAT^L})|_{fAT^L} \mathcal{D}[R_x]) \mid R_x \in Fba(E[x/it])\}$, is directed in AT^L . Thus G is directed in $(EOT \rightarrow AT^L)$ and $\{\mathcal{D}[R] \mid R \in Fba(\mathbf{read}(t).E)\}$ is directed in AT^L .

Since in_{AT^L} is continuous in its second argument, we have

$$\begin{aligned} & \sqcup \{\mathcal{D}[R] \mid R \in Fba(\mathbf{read}(t).E)\} \\ &= \sqcup \{in_{AT^L}(\wp(it), g) \mid it = \mathcal{J}[t], g \in G\} \\ &= in_{AT^L}(\wp(it), \sqcup G) \end{aligned}$$

On the other hand, by inductive hypothesis and continuity of the operators on AT^L ,

$$\begin{aligned} & \mathcal{D}[\mathbf{read}(t).E] \\ &= in_{AT^L}(\wp(it), \lambda x.[match(it, x)] \rightarrow (out_{AT^L}^x(nil_{AT^L})|_{AT^L} \mathcal{D}[E[x/it]]), \perp) \\ &= in_{AT^L}(\wp(it), \lambda x.[match(it, x)] \rightarrow \\ & \quad (out_{AT^L}^x(nil_{AT^L})|_{AT^L} (\sqcup \{\mathcal{D}[R_x] \mid R_x \in Fba(E[x/it])\})), \perp) \\ &= in_{AT^L}(\wp(it), \sqcup K) \end{aligned}$$

where

$$\begin{aligned} K &= \{f : EOT \rightarrow AT^L \mid f \equiv (\lambda x.[match(it, x)] \rightarrow (out_{AT^L}^x(nil_{AT^L})|_{AT^L} \mathcal{D}[R_x]), \perp), \\ & \quad \forall x \in \mathcal{MT}(\{\wp(it)\}) : R_x \in Fba(E[x/it])\}. \end{aligned}$$

We are only left to show that $\sqcup G = \sqcup K$. Since $G \subseteq K$ then obviously $\sqcup G \leq_{AT^L} \sqcup K$. On the other hand, let $f \in K$ and consider the chain of functions $\{g_j \mid j \geq 0\} \subseteq H$ with, $\forall j \geq 0$, g_j defined by $g_j = \lambda x.[x \in M_j] \rightarrow f(x), \perp$ where $M_0 \subseteq M_1 \subseteq \dots$ is a chain of finite subsets of EOT such that $\bigcup_j M_j = \mathcal{MT}(\{\wp(it)\})$. It is easily seen that $\sqcup \{g_j \mid j \geq 0\} = f$. We can now deduce that $\sqcup K \leq_{AT^L} \sqcup G$, and then $\sqcup G = \sqcup K$, by applying the following fact from the theory of cpo's: given X and Y subsets of a cpo D with $s_x = \sqcup X$ and $s_y = \sqcup Y$, if for each $y \in Y$ there exists $A_y \subseteq X$ such that $y = \sqcup A_y$ then $s_y \leq_D s_x$.

3. The proof proceeds by induction on the depth of the proof of $R \sqsubseteq_{\mathcal{RP}} P$ within the proof system \mathcal{RP} . Here, we consider the case when III is the last applied rule.

If III is the last applied rule for deducing $R \sqsubseteq_{\mathcal{RP}} P$ then, since R is compact, there exist $it \in EIT$, $M \subseteq_{fin} \mathcal{MT}(\{\wp(it)\})$, E and F finite such that $fv(E) \subseteq var(it)$, $fv(F) \subseteq var(it)$, $R =_{\mathcal{RP}} \mathbf{in}(it).E$, $P =_{\mathcal{RP}} \mathbf{in}(it).F$, $\forall ot \in \mathcal{MT}(\{\wp(it)\}) \setminus M : E[ot/it] =_{\mathcal{RP}} \Omega$ and $\forall ot \in M : E[ot/it] \sqsubseteq_{\mathcal{RP}} F[ot/it]$. Since $\forall ot \in M : E[ot/it]$ is compact, by induction we may assume that $\forall ot \in M, \exists R_{ot} \in Fba(F[ot/it]) : E[ot/it] \sqsubseteq_{\mathcal{RP}} R_{ot}$. Let us now consider process $R'' \equiv \mathbf{in}(it).Q(M)$ (where $Q(M)$ is the process introduced in Definition 7.9(6)). By definition, it follows that $R'' \in Fba(\mathbf{in}(it).F)$. By applying rule III we deduce that $R \sqsubseteq_{\mathcal{RP}} R''$. Since $R'' \in Fba(\mathbf{in}(it).F)$ and $P =_{\mathcal{RP}} \mathbf{in}(it).F$ imply that there exists $R' \in Fba(P)$ such that $R'' =_{\mathcal{RP}} R'$, we conclude that $R \sqsubseteq_{\mathcal{RP}} R'$.

4. Suppose that $\forall R \in Fba(P): R \sqsubseteq_{\mathcal{RP}} Q$ and that $P \text{ must } O$. By paralleling the proof of Proposition 6.21, we can prove that for any process P and observer O , if $P \text{ must } O$ then there exists a compact R such that $R \text{ must } O$ and $R \sqsubseteq_{\mathcal{RP}} P$. Since R is compact and $R \sqsubseteq_{\mathcal{RP}} P$ then from part 3 of this proposition we deduce that there exists $R' \in Fba(P)$ such that $R \sqsubseteq_{\mathcal{RP}} R'$. By hypothesis, we have that $R' \sqsubseteq_{\mathcal{RP}} Q$. Therefore, $R \text{ must } O$ implies $Q \text{ must } O$. \square

To prove full abstraction for compact processes (Proposition 7.17), we shall use an alternative characterization of \sqsubseteq_{fAT^L} .

Definition 7.11. Let $T, U \in fAT^L$, $\alpha \in \mathcal{Act}$ and $\rho \in \mathcal{Act}^*$.

- We write $T \xrightarrow{\alpha} U$ if one of the following conditions holds:
 - $\alpha = ot!$, $T = \langle \mathcal{A}, f_{ir} \uplus f_{out} \rangle$, $(o, ot) \in Ev_{out}(\mathcal{A})$ and $U = f_{out}(ot)$;
 - $\alpha = ot?$, $T = \langle \mathcal{A}, f_{ir} \uplus f_{out} \rangle$, $\exists it: match(it, ot) \wedge (i, \wp(it)) \in Ev_{ir}(\mathcal{A})$ and $U = f_{ir}(ot)$.
- The *acceptance set* of T after ρ is
 - 1.

$$\mathcal{A}(T, \varepsilon) = \begin{cases} \mathcal{A} & \text{if } T = \langle \mathcal{A}, f_{ir} \uplus f_{out} \rangle \\ \emptyset & \text{otherwise (i.e. if } T = \perp) \end{cases}$$

2.

$$\mathcal{A}(T, \alpha \cdot \rho') = \begin{cases} \mathcal{A}(U, \rho') & \text{if } T \xrightarrow{\alpha} U \\ \emptyset & \text{otherwise} \end{cases}$$

- We write
 1. $T \downarrow \varepsilon$ if $T \neq \perp$,
 2. $T \downarrow \alpha \cdot \rho'$ if $T \downarrow \varepsilon$ and $T \xrightarrow{\alpha} U$ implies $U \downarrow \rho'$.
- We write $T \ll_{fAT^L} U$ if for every $\rho' \in \mathcal{Act}^*$

$$T \downarrow \rho' \Rightarrow \begin{cases} 1. U \downarrow \rho' \\ 2. \mathcal{A}(U, \rho') \subset \subset \mathcal{A}(T, \rho'). \end{cases}$$

Both the propositions below can be proven like analogous results in [35].

Proposition 7.12. For $T, U \in fAT^L$, if $T \ll_{fAT^L} U$ then $T \downarrow \alpha$ and $U \xrightarrow{\alpha} U'$ imply that there exists T' such that $T \xrightarrow{\alpha} T'$ and $T' \ll_{fAT^L} U'$.

Proposition 7.13. For $T, U \in AT^L$, $T \leq_{fAT^L} U$ if and only if $T \ll_{fAT^L} U$.

The following results will be used for proving full abstraction for compact processes. To prove them we will use soundness of the reduced proof system \mathcal{RP} with respect to both \ll_M and \ll_{fAT^L} (soundness of \mathcal{RP} w.r.t. \ll_{fAT^L} can be proven by paralleling the proof of soundness of \mathcal{RP} w.r.t. \sqsubseteq_M), the notion of *hnf* and the following property about *hnfs*.

Proposition 7.14. *For any process P such that $P \downarrow \varepsilon$ and for every action $\alpha \in \mathcal{Act}$: $h(P) \Rightarrow \xrightarrow{\alpha} P'$ iff $\mathcal{D}[P] \xrightarrow{\alpha} \mathcal{D}[P']$.*

Proof. It directly follows by construction of *hnfs* and by definitions of $\mathcal{D}[\cdot]$ and of the partial function $\xrightarrow{\alpha}$ on fAT^L . \square

Proposition 7.15. *For every compact process R and every $\rho \in \mathcal{Act}^*$, $R \downarrow \rho$ iff $\mathcal{D}[R] \downarrow \rho$.*

Proof. The proof proceeds by induction on $|\rho|$, and the length of ρ .

If $|\rho| = 0$, i.e. $\rho = \varepsilon$, and $R \downarrow \varepsilon$ then R has a *hnf*, $h(R)$, and since the proof system \mathcal{RP} is sound with respect to \ll_{fAT^L} we have $\mathcal{D}[R] = \mathcal{D}[h(R)]$. Therefore, since *hnfs* are interpreted as non-trivial trees we have $\mathcal{D}[R] \neq \perp$. Conversely, since R is compact, $\mathcal{D}[R] \neq \perp$ implies $R \downarrow \varepsilon$.

Let us assume now that $\rho = \alpha \cdot \rho'$. If $R \downarrow \varepsilon$ then R has a *hnf*, $h(R)$ such that $R =_{\mathcal{RP}} h(R)$. Let R_x be such that $h(R) \Rightarrow \xrightarrow{\alpha} R_x$. By Proposition 7.14, we have $\mathcal{D}[h(R)] \xrightarrow{\alpha} \mathcal{D}[R_x]$. The hypothesis $R \downarrow \rho$ implies $R_x \downarrow \rho'$, hence, by induction, we may assume that $\mathcal{D}[R_x] \downarrow \rho'$. Therefore, $\mathcal{D}[h(R)] \downarrow \alpha \cdot \rho'$. Since $\mathcal{D}[R] = \mathcal{D}[h(R)]$ we conclude that $\mathcal{D}[R] \downarrow \alpha \cdot \rho'$. Conversely, suppose that $\mathcal{D}[R] \downarrow \alpha \cdot \rho'$, that is (by definition) $\mathcal{D}[R] \xrightarrow{\alpha} T$ implies $T \downarrow \rho'$. From the case $|\rho| = 0$, we already know that the result is true for $\rho = \varepsilon$ then $R \downarrow \varepsilon$ which implies that there exists a *hnf*, $h(R)$ such that $R =_{\mathcal{RP}} h(R)$. By Proposition 7.14, we deduce that $h(R) \Rightarrow \xrightarrow{\alpha} R_x$ and $T = \mathcal{D}[R_x]$. Since $T \downarrow \rho'$, by induction, we can assume that $R_x \downarrow \rho'$. Since this holds for all R_x such that $R \xrightarrow{\alpha} R_x$ then, by definition, $R \downarrow \alpha \cdot \rho'$. \square

Proposition 7.16. *For R compact process and $\rho \in \mathcal{Act}^*$, $R \downarrow \rho$ implies $\text{sat}(\mathcal{A}(R, \rho)) = \mathcal{A}(\mathcal{D}[R], \rho)$.*

Proof. The proof proceeds by induction on ρ . Let us assume that $R \downarrow \rho$. This implies that R has a *hnf*, $h(R)$ and $R =_{\mathcal{RP}} h(R)$ and $\mathcal{D}[R] = \mathcal{D}[h(R)]$.

Let $\rho = \varepsilon$. Since $R =_{\mathcal{RP}} h(R)$ then $R \ll_{\mathcal{M}} h(R)$ and $h(R) \ll_{\mathcal{M}} R$. This means that $\text{sat}(\mathcal{A}(R, \varepsilon)) = \text{sat}(\mathcal{A}(h(R), \varepsilon))$. By definition of *hnfs*, we have $\text{sat}(\mathcal{A}(h(R), \varepsilon)) = \mathcal{A}(h(R), \varepsilon)$. By the construction of *hnfs*, we have $\mathcal{A}(h(R), \varepsilon) = \mathcal{A}(\mathcal{D}[h(R)], \varepsilon)$. Since $\mathcal{D}[h(R)] = \mathcal{D}[R]$ we conclude that $\text{sat}(\mathcal{A}(R, \varepsilon)) = \mathcal{A}(\mathcal{D}[R], \varepsilon)$.

Let $\rho = \alpha \cdot \rho'$. By induction, if either of $\mathcal{A}(R, \alpha \cdot \rho')$ or $\mathcal{A}(\mathcal{D}[R], \alpha \cdot \rho')$ is non-empty, then both are non-empty. Let us assume that they both are non-empty. By definition, $\text{sat}(\mathcal{A}(R, \alpha \cdot \rho')) = \text{sat}(\bigcup \{ \mathcal{A}(R_x, \rho') \mid R \xrightarrow{\alpha} R_x \})$. If R' is such that $h(R) \Rightarrow \xrightarrow{\alpha} R'$ then $\text{sat}(\bigcup \{ \mathcal{A}(R_x, \rho') \mid R \xrightarrow{\alpha} R_x \}) = \text{sat}(\mathcal{A}(R', \rho'))$. By induction we can assume that $\text{sat}(\mathcal{A}(R', \rho')) = \mathcal{A}(\mathcal{D}[R'], \rho')$. Therefore, by Proposition 7.14, $\mathcal{A}(\mathcal{D}[R'], \rho') = \mathcal{A}(\mathcal{D}[h(R)], \alpha \cdot \rho')$. Finally, since $\mathcal{D}[h(R)] = \mathcal{D}[R]$, we have $\text{sat}(\mathcal{A}(R, \alpha \cdot \rho')) = \mathcal{A}(\mathcal{D}[R], \alpha \cdot \rho')$. \square

Proposition 7.17. *For compact processes R and R' , $R \sqsubseteq_{\mathcal{RP}} R'$ iff $\mathcal{D}[R] \leq_{fAT^L} \mathcal{D}[R']$.*

Proof. Since $R \sqsubseteq_{\mathcal{CP}} R'$ iff $R \sqsubseteq_M R'$ iff $R \ll_M R'$ (by completeness of \mathcal{CP}), and $T \leq_{fAT^L} U$ iff $T \ll_{fAT^L} U$, for all compact acceptance trees T and U , then, by Proposition 7.10(1), it suffices to show that $R \ll_M R'$ iff $\mathcal{D}[R] \ll_{fAT^L} \mathcal{D}[R']$. This directly follows from Propositions 7.15 and 7.16. \square

We now consider finite processes.

Theorem 7.18. *For finite processes P and Q , $P \sqsubseteq_{\mathcal{RP}} Q$ iff $\mathcal{D}[P] \leq_{AT^L} \mathcal{D}[Q]$.*

Proof. (\Rightarrow) Suppose that $P \sqsubseteq_{\mathcal{RP}} Q$. By Proposition 7.10(3), we have that for each $R \in Fba(P)$ there exists $R' \in Fba(Q)$ such that $R \sqsubseteq_{\mathcal{RP}} R'$. By Proposition 7.17, this implies that $\mathcal{D}[R] \leq_{fAT^L} \mathcal{D}[R']$. Therefore, by Proposition 7.10(2), we have $\mathcal{D}[P] = \bigsqcup \{\mathcal{D}[R] \mid R \in Fba(P)\} \leq_{AT^L} \bigsqcup \{\mathcal{D}[R'] \mid R' \in Fba(Q)\} = \mathcal{D}[Q]$.

(\Leftarrow) Suppose that $\mathcal{D}[P] \leq_{AT^L} \mathcal{D}[Q]$. By Proposition 7.10(2), this implies that $\forall R \in Fba(P) : \mathcal{D}[R] \leq_{AT^L} \mathcal{D}[Q]$. For each $R \in Fba(P)$, since $\{\mathcal{D}[R'] \mid R' \in Fba(Q)\}$ is directed in AT^L (Proposition 7.10(2)) and $\mathcal{D}[R]$ is compact in AT^L (Proposition 7.10(1)), there exists $R' \in Fba(Q)$ such that $\mathcal{D}[R] \leq_{AT^L} \mathcal{D}[R']$. By Proposition 7.17, we have $R \sqsubseteq_{\mathcal{RP}} R'$. By definition, $R' \in Fba(Q)$ implies $R' \sqsubseteq_{\mathcal{RP}} Q$ and thus $R \sqsubseteq_{\mathcal{RP}} Q$. By Proposition 7.10(4), we conclude that $P \sqsubseteq_{\mathcal{RP}} Q$. \square

Full abstraction for general processes is proven by using finite-breadth approximants.

Definition 7.19. For each non-finite process P , $Fba(P) = \{R \mid \exists n \geq 0 : R \in Fba(P^n)\}$.

Theorem 7.20. *For all processes P and Q , $P \sqsubseteq_{\mathcal{CP}} Q$ iff $\mathcal{D}[P] \leq_{AT^L} \mathcal{D}[Q]$.*

Proof. For every term E and every $\xi \in Env_{AT^L}$, we have

1. $\mathcal{D}[E^n]\xi \leq_{AT^L} \mathcal{D}[E^{n+1}]\xi$, for every $n \geq 0$,
2. $\mathcal{D}[E]\xi = \bigsqcup \{\mathcal{D}[E^n]\xi \mid n \geq 0\}$.

This is a standard result in the algebraic semantics theory and could be proven by structural induction, paralleling the corresponding proofs (e.g. that of Theorem 4.2.11) in [28]. From the previous result, Proposition 7.10 and Theorem 7.18, it directly follows that

$$\{\mathcal{D}[R] \mid R \in Fba(P)\} \text{ is directed in } AT^L \text{ and } \mathcal{D}[P] = \bigsqcup \{\mathcal{D}[R] \mid R \in Fba(P)\}. \quad (2)$$

Moreover, for all processes P and Q , $P \sqsubseteq_{\mathcal{CP}} Q$ iff $\forall n \geq 0, \exists m \geq 0 : P^n \sqsubseteq_{\mathcal{CP}} Q^m$ (this is a standard result for type systems similar to us). Hence, from Proposition 7.10 it follows that

$$P \sqsubseteq_{\mathcal{CP}} Q \text{ iff } \forall R \in Fba(P), \exists R' \in Fba(Q) : R \sqsubseteq_{\mathcal{CP}} R'. \quad (3)$$

Since $\{\mathcal{D}[R] \mid R \in Fba(P)\}$ is a set of compact elements in AT^L , from (2) it follows that

$$\mathcal{D}[P] \leq_{AT^L} \mathcal{D}[Q] \text{ iff } \forall R \in Fba(P), \exists R' \in Fba(Q) : \mathcal{D}[R] \leq_{AT^L} \mathcal{D}[R'].$$

Since it has been already proven that $R \sqsubseteq_{\mathcal{CP}} R'$ iff $\mathcal{D}[R] \leq_{ATL} \mathcal{D}[R']$, for compact processes R and R' , from (3) we conclude that $P \sqsubseteq_{\mathcal{CP}} Q$ iff $\mathcal{D}[P] \leq_{ATL} \mathcal{D}[Q]$. \square

8. IPAL: imperative PAL

In this section, we will show that the framework we have defined for PAL can easily accommodate the addition of an imperative construct (in the form of action prefixing) to the language. In particular, we show that the theory developed for PAL can be reused to establish that also the proof system for IPAL is sound and complete (and the denotational model is fully abstract).

We assume that each individual process has its own private store (for binding variables to values), which can be accessed by other processes only by explicit communications. However, for assigning values to free occurrences of variables, we take advantage of the syntactic restriction of Definition 3.1 about (value and process) variables binders and, differently from [30,23], we do not explicitly model the store but use explicit substitutions. This choice allows us to smoothly extend the framework for PAL to its imperative variant IPAL. For example, the states of the LTS that characterizes the operational semantics of the language are purely syntactical objects like in PAL and we avoid considering configurations (i.e. pairs of processes and stores) and operators over them.

The syntax of IPAL is obtained by adding a new prefixing operator for assignment to that of PAL (Definition 3.1). Thus, the productions for IPAL action prefixes are

$$a ::= \text{out}(t) \mid \text{in}(t) \mid \text{read}(t) \mid \text{eval}(E) \mid x := e$$

Obviously, the unary operator $x := e$ binds the variable x within its argument term and is a new binder for value-variables.

The operational semantics of IPAL is characterized via a LTS which is obtained by adding to the LTS for PAL the following rule:

$$\text{IR14} \quad x := e.E \rightarrow E[\mathcal{E}[e]/x]$$

which accounts for the behaviour of assignment prefixes. Rule IR14 models an internal move that updates the store. It affects only the argument of the prefixing but has no effect on parallel processes which have free occurrences of variables with the same name of the variable on the left of $:=$. This change is not directly observable, only explicit communications of the environment via an out operation make it evident.

Observers cannot access the private store of tested processes, but can only gather information by communication. The behavioural preorders \sqsubseteq_M and \ll_M are defined as for PAL and their coincidence (Theorem 5.13) can be proven again.

A proof system for IPAL is obtained by simply adding a specific law for assignment

$$\text{AS} \quad x := e.X = X[e/x]$$

to the proof system for PAL. Since stores can only be investigated by communication or by conditional choice, we do not introduce any additional inference rule for ensuring substitutivity of value expressions (rules VII and VIII in Table 10 are sufficient).

All of the results related to the equational semantics of PAL can be proven for IPAL as well. In particular, in the normalization procedures, law AS is employed for removing (leading) assignments. By paralleling the proofs given for PAL, it is easy to check that the proof system for IPAL is sound and complete with respect to the testing preorders (Theorem 6.23).

For defining the denotational semantics of IPAL, like for the **eval** and the conditional operators, we do not use a specific operator on fAT^L for assignment prefixes. Indeed, we only add the following clause to the definition of the interpretation function $\mathcal{D}[\cdot]$ given in Section 7:

$$10. \quad \mathcal{D}[x := e.E]\xi = \mathcal{D}[E[\mathcal{D}[e]/x]]\xi.$$

The finite-breath approximants of a finite IPAL process of the form $x := e.E$ are given by

$$7. \quad Fba(x := e.E) = \{Q \mid Q \in Fba(E[\mathcal{D}[e]/x])\}.$$

Again, all of the results concerning the denotational semantics of PAL hold for IPAL. Hence, the denotational model is still fully abstract with respect to both the behavioural preorders and the proof-theoretic one (Theorem 7.20).

9. Conclusions and related work

In this paper we have studied the impact of a theory of testing of [19] on two process description languages that permit writing programs that manipulate values and exchange them asynchronously with other programs. The two languages are obtained by substituting the uninterpreted actions of a CSP-like process algebra with the Linda primitives for process interaction (PAL) and by adding to PAL an assignment command (IPAL). Sound and complete proof systems for testing have been defined together with a fully abstract denotational model that is based on natural interpretations. This work has been instrumental for the development of KLAIM, a programming language based on PAL for implementing interactive and mobile agents [18].

Asynchronous variants of process algebras have been already considered in the literature for ACP [7], CSP [32], π -calculus [38] and CCS [37]. These works have followed two main lines that differ for the way non-blocking output actions are modelled: They are rendered either as state transformers or as processes.

The variants of ACP [8,9], CCS [17] and CSP [36] model output actions as state transformers: They associate buffers (modelled as state operators in ACP and CCS, and as processes in CSP) to channels. These variants naturally describe systems with outputs modelled as unblocked sending primitives that make messages available for consumption.

In [8,9], asynchronous sending operations are visible. A consequence of this is that processes which only differ for the sending order of messages are considered as different. An “ad hoc” notion of failure equivalence had to be introduced to correctly describe process behaviours with respect to deadlock. Here, we can use the usual testing scenario of synchronous process algebras and exploit the different observation (i.e. communication) mechanism to obtain a different semantics.

In [17], sending operations are not visible and an auxiliary operator is used to store the messages that are sent by processes. Thus, messages are somehow linked to the sender process and cannot be read by it. This approach is not suitable to model the Linda communication paradigm.

In [36], it is shown that CSP processes with asynchronous communications can be obtained by attaching a buffer to each of the input and output channels of CSP processes. This scenario introduces a (centralized) manager process for each communication channel and asynchrony strongly relies on the fact that sending messages to channel managers is always possible. Our point of view is that asynchronous communications are more realistic assumptions for distributed systems; thus we model them as (first-class) language primitives.

The variants of π -calculus [33,34,11,26,2] and that of CCS described in [42] model output actions as processes, and use bisimulation-based equivalences to obtain observational semantics. We have followed a similar approach; output actions are modelled by means of internal moves that can always take place (i.e. are non-blocking) and cannot change the structure of terms. This choice, in particular, implies that

$$\mathbf{out}(t_1).(\mathbf{out}(t_2).\mathbf{nil}[]\mathbf{out}(t_3).\mathbf{nil}) \neq (\mathbf{out}(t_1).\mathbf{out}(t_2).\mathbf{nil})[](\mathbf{out}(t_1).\mathbf{out}(t_3).\mathbf{nil}),$$

which is apparently in contrast with [10], where the law

$$\bar{a}.(\bar{b}.\mathbf{nil} + \bar{c}.\mathbf{nil}) = \bar{a}.\bar{b}.\mathbf{nil} + \bar{a}.\bar{c}.\mathbf{nil} \quad (4)$$

(\bar{a} , \bar{b} and \bar{c} denote outputs on channels a , b and c) is considered an essential law for models of asynchronous communications. Actually, the difference is due only to the distinct choice operators of the languages. Indeed, the $+$ operator used in [10] can be used to describe both internal and external non-determinism. For example, with the term $\bar{a}.b.\mathbf{nil} + c.\mathbf{nil}$ (b and c denote inputs on channels b and c) the sending of a would permit rejecting of c . In our setting, the corresponding term would be $(\bar{a}.b.\mathbf{nil}[]c.\mathbf{nil}) \oplus \bar{a}.b.\mathbf{nil}$. Output actions are dealt with just like the silent action of CCS in the translation from CCS to TCCS of [20]. Therefore, in our setting the sound version of (4) above is

$$\begin{aligned} & \mathbf{out}(t_1).((\mathbf{out}(t_2).\mathbf{nil}[]\mathbf{out}(t_3).\mathbf{nil}) \oplus \mathbf{out}(t_2).\mathbf{nil} \oplus \mathbf{out}(t_3).\mathbf{nil}) \\ &= (\mathbf{out}(t_1).\mathbf{out}(t_2).\mathbf{nil}[]\mathbf{out}(t_1).\mathbf{out}(t_3).\mathbf{nil}) \\ & \oplus (\mathbf{out}(t_1).\mathbf{out}(t_2).\mathbf{nil}) \oplus (\mathbf{out}(t_1).\mathbf{out}(t_3).\mathbf{nil})). \end{aligned}$$

Only a few well-established theories for process calculi which explicitly manipulate values have been developed. The only addition to [8,9], that we have already mentioned, are [29,30,35]. There a testing framework is developed for a variant of TCCS

[20,28] with value-passing. By and large, we have used methods similar to those of [29,30,35]; however, tuples-based asynchronous communication calls for a different formal setup. Apart for the presence of non-finitely branching transition systems, we had to face additional complications introduced by the inability of observers to perceive the differences among patterns which access a given tuple. To take this problem into account, we introduced the notion of (closed set of) events. The major impact of the communication mechanism on the denotational model is that the sequel of an input action is a partial function defined only for the tuples that match the pattern used.

Another approach to the formal analysis of the semantics of Linda-based communication paradigms has been followed in [15]. There, one can find another example of tuning the process algebraic techniques for dealing with the Linda paradigm. The basic idea is that of considering tuples as atomic items with a unique identification name. This choice, on the one hand, simplifies the required mathematics, but on the other, prevents taking into account all of the subtleties of the Linda communication model. A similar behaviour can also be modelled within our framework by introducing the simplifying assumption that tuples are atomic items.

Our extension to IPAL of the semantic set-up for PAL is simpler than that of [30,35] for CCS with value-passing and assignment. There, stores (for bindings variables to values) are explicitly modelled, and the operational semantics has to consider configurations (i.e. pairs of processes and stores) and operators over them. Moreover, the new proof system is obtained by extending the (applicative) laws for PAL with a family of laws (one for each process operator) for rewriting assignment in the normalization procedure and an inference rule for ensuring substitutivity of expressions in assignments. Here, by taking advantage of the syntactic restriction of Definition 3.1 about (value and process) variables binders, we show that a single additional law is sufficient for the complete equational characterization of IPAL. Had we removed the syntactic restriction, we would have to use parameterized process variables in order to avoid providing PAL with a counterintuitive and unsatisfactory semantics that models recursive terms differently from their unfoldings.

The use of action prefixing instead of full sequential composition has also been essential for “reusing” the semantical machinery introduced in [22,43] for PAL. Had we chosen to use sequential composition, terms would inherit stores and the equivalences would not be congruences.

The development of a similar framework to deal with full sequential program composition rather than with action prefixing is under progress. In [21] we have already studied an imperative language, *L*, obtained by embedding the Linda primitives for interprocess communication in a simple imperative language with sequential composition. We succeeded in defining a testing scenario for *L*, by enabling observers to test the (final) store of (finite computations of) programs; but we were not able to obtain an equational characterizations of the testing preorders over this richer language. Obviously, this makes it difficult to use that framework for verifying programs.

Additional work is needed also to deal properly with the (left and communication) merge operators and the (general) external choice operator. Their use, on the one hand,

has rendered the definition of the alternative behavioural characterization and of the proof system easier; but, on the other, it has significantly increased the discriminating power of observers. Indeed, the merge operators permit expressing causal dependencies on output actions. Thus, our observation mechanism allows observers to determine whether a system has actually consumed a message. Moreover, outputs at choice points require synchronizations at the implementation level. All this may conflict with the idea that asynchronous outputs are intended to take place immediately without requiring availability of a corresponding input; in these circumstances it might be argued that observers cannot be guaranteed that a message has been consumed. As a consequence we have that our observational theory is, to a certain extent, too discriminating; indeed, some equational laws for asynchronous bisimulation of [2] are not valid for our testing equivalence. We see two possibilities for weakening our behavioural relations in this respect: omitting the merge operators and using a less general (input guarded) external choice operator or modifying the observation mechanism.

Acknowledgements

We are grateful to Michele Boreale and GianLuigi Ferrari for helpful comments and discussions, and to Anna Ingólfssdóttir and the two anonymous referees for the insightful comments that have allowed us to significantly improve the paper. This work has also benefitted from comments by Ernst-Rüdiger Olderog and Catuscia Palamidessi, the external referees of the Ph.D. thesis of the second author on which the paper is based.

References

- [1] L. Aceto, A. Ingólfssdóttir, A theory of testing for ACP, in: J.C.M. Baeten, J.F. Groote (Eds.), CONCUR'91, Proc. Lecture Notes in Computer Science, Vol. 527, Springer, Berlin, 1991, pp. 78–95.
- [2] R.M. Amadio, I. Castellani, D. Sangiorgi, On bisimulations for the asynchronous π -calculus, Theoret. Comput. Sci. 195 (2) (1998) 291–324.
- [3] A. Brogi, P. Ciancarini, The concurrent language shared prolog, ACM Trans. Programming Languages Systems 13 (1) (1991) 99–123.
- [4] M. Boreale, R. De Nicola, Testing equivalence for mobile processes, Inform. and Comput. 120 (2) (1995) 279–303.
- [5] S.D. Brookes, C.A.R. Hoare, A.W. Roscoe, A theory of communicating sequential processes, J. ACM 31 (3) (1984) 560–599.
- [6] L. Borrmann, M. Herdierckhoff, A. Klein, Tuple space integrated into modula-2: implementation of the Linda concept on a hierarchical multiprocessor, in: R. Jesshope (Ed.), CONPAR'88, Proc. Cambridge University Press, Cambridge, 1988.
- [7] J. Bergstra, J.W. Klop, Process algebra for synchronous communication, Inform. Control 60 (1984) 109–137.
- [8] J.A. Bergstra, J.W. Klop, J.V. Tucker, Process algebra with asynchronous communication mechanisms, in: S.D. Brookes, A.W. Roscoe, G. Winskel (Eds.), Seminar in Concurrency, Proc. Lecture Notes in Computer Science, Vol. 197, Springer, Berlin, 1985, pp. 76–95.
- [9] F.S. de Boer, J.W. Klop, C. Palamidessi, Asynchronous communication in process algebra, Proc. LICS, IEEE Computer Society Press, silver Spring, MD, 1992, pp. 137–147.

- [10] F.S. de Boer, C. Palamidessi, On the asynchronous nature of communication in concurrent logic languages: a fully abstract model based on sequences, in: J.C.M. Baeten, J.W. Klop (Eds.), CONCUR'90, Proc., Lecture Notes in Computer Science, Vol. 458, Springer, Berlin, 1990, pp. 99–114.
- [11] G. Boudol, Asynchrony in the π -calculus, Research Report 1702, INRIA Sophia-Antipolis, 1992.
- [12] N. Carriero, D. Gelernter, Linda in context, *Comm. ACM* 32 (4) (1989) 444–458.
- [13] N. Carriero, D. Gelernter, J. Leichter, Distributed data structures in Linda, *Proc. ACM Symp. on Principles of Programming Languages*, ACM, New York, 1986, pp. 236–242.
- [14] K.M. Chandy, J. Misra, *Parallel Program Design: A Foundation*, Addison-Wesley, MA, 1988.
- [15] P. Ciancarini, R. Gorrieri, G. Zavattaro, Towards a calculus for generative communication, *Proc. 1st IFIP Conf. on Formal Methods for Open Object-Based Distributed Systems*, FMOODS'96, Chapman & Hall, London, 1996, pp. 283–297.
- [16] R. Cleaveland, M.C.B. Hennessy, Testing equivalence as a bisimulation equivalence, *J. Formal Aspects Comput. Sci.* 5 (1993) 1–20.
- [17] R. Cleaveland, D. Yankelevich, An operational framework for value-passing processes, *Proc. ACM Symp. on Principles of Programming Languages*, ACM, New York, 1994, pp. 326–338.
- [18] R. De Nicola, G.-L. Ferrari, R. Pugliese, KLAIM: a Kernel language for agents interaction and mobility, *IEEE Trans. Software Engineering* 24 (5) (1998) 315–330.
- [19] R. De Nicola, M.C.B. Hennessy, Testing equivalence for processes, *Theoret. Comput. Sci.* 34 (1984) 83–133.
- [20] R. De Nicola, M.C.B. Hennessy, CCS without τ 's, in: H. Ehrig, R. Kowalsky, G. Levi, U. Montanari (Eds.), TAPSOFT'87, *Proc. Lecture Notes in Computer Science*, Vol. 249, Springer, Berlin, 1987, pp. 138–152.
- [21] R. De Nicola, R. Pugliese, An observational semantics for Linda, in: J. Desel (Ed.), STRICT'95, *Proc. Series Workshops in Computing*, Springer, Berlin, 1995, pp. 129–143.
- [22] R. De Nicola, R. Pugliese, A process algebra based on Linda, in: P. Ciancarini, C. Hankin (Eds.), COORDINATION'96, *Proc. Lecture Notes in Computer Science*, Vol. 1061, Springer, Berlin, 1996, pp. 160–178.
- [23] R. De Nicola, R. Pugliese, Testing semantics of asynchronous distributed programs, in: M. Dam (Ed.), *Analysis and Verification of Multiple-Agent Languages*, *Proc. Lecture Notes in Computer Science*, Vol. 1192, Springer, Berlin, 1996, pp. 320–345.
- [24] D. Gelernter, Generative communication in Linda, *ACM Trans. Programming Languages Systems* 7 (1) (1985) 80–112.
- [25] I. Guessarian, *Algebraic Semantics*, *Lecture Notes in Computer Science*, Vol. 99, Springer, Berlin, 1981.
- [26] M. Hansen, H. Huttel, J. Kleist, Bisimulations for asynchronous mobile processes. *Proc. Tbilisi Symp. on Languages, Logic and Computation 1995*, also appeared as Research Paper HCRC/RP-72, Univ. of Edinburgh.
- [27] M.C.B. Hennessy, Acceptance trees, *J. ACM* 32 (4) (1985) 896–928.
- [28] M.C.B. Hennessy, *Algebraic Theory of Processes*, The MIT Press, Cambridge, MA, 1988.
- [29] M.C.B. Hennessy, A. Ingolfssdottir, A theory of communicating processes with value-passing, *Inform. and Comput.* 107 (2) (1993) 202–236.
- [30] M.C.B. Hennessy, A. Ingolfssdottir, Communicating processes with value-passing and assignment, *J. Formal Aspects of Comput. Sci.* 3 (1993) 346–366.
- [31] M.C.B. Hennessy, G.D. Plotkin, A term model for CCS, in: P. Dembinsky (Ed.), MFCS'80, *Proc., Lecture Notes in Computer Science*, Vol. 88, Springer, Berlin, 1980, pp. 261–274.
- [32] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, Englewood Cliffs, NJ, 1985.
- [33] K. Honda, M. Tokoro, An object calculus for asynchronous communication, in: P. America (Ed.), ECOOP'91, *Proc., Lecture Notes in Computer Science*, Vol. 512, Springer, Berlin, 1991, pp. 133–147.
- [34] K. Honda, M. Tokoro, On asynchronous communication semantics, in: M. Tokoro, O. Nierstrasz, P. Wegner (Eds.), *Object-based Concurrent Computing*, *Lecture Notes in Computer Science*, Vol. 612, Springer, Berlin, 1991, pp. 21–51.
- [35] A. Ingolfssdottir, Semantic models for communicating processes with value-passing, Ph.D. Thesis, University of Sussex, Department of Cognitive and Computing Science, 1994.
- [36] H. Jifeng, M.B. Josephs, C.A.R. Hoare, A theory of synchrony and asynchrony, in: M. Broy, C.B. Jones (Eds.), *IFIP Working Conf. on Programming Concepts and Methods*, *Proceedings*, North-Holland, Elsevier, Amsterdam, 1990, pp. 459–478.

- [37] R. Milner, *Communication and Concurrency*, Prentice-Hall, Englewood Cliffs, NJ, 1989.
- [38] R. Milner, J. Parrow, D. Walker, A calculus of mobile processes, Part I and II, *Inform. and Comput.* 100 (1992) 1–77.
- [39] D. Park, Concurrency and automata on infinite sequences, in: P. Deussen (Ed.), *Theoretical Computer Science, Lecture Notes in Computer Science*, Vol. 104, Springer, Berlin, 1980, pp. 167–183.
- [40] J. Pinakis, C. McDonald, The inclusion of the Linda Tuple space operations in a Pascal-based concurrent language, *Computer Science Dept., Univ. of Western Australia*, 1991.
- [41] G.D. Plotkin. A structural approach to operational semantics, Technical Report DAIMI FN-19, Aarhus University, Dept. of Computer Science, Denmark, 1981.
- [42] R. Pugliese, A process calculus with asynchronous communications, in: A. De Santis (Ed.), *Proc. 5th Italian Conf. on Theoretical Computer Science '95*, World Scientific, Singapore, 1995.
- [43] R. Pugliese, Semantic theories for asynchronous languages, Ph.D. Thesis, University of Rome “La Sapienza”, Dipartimento di Scienze dell’Informazione, 1996.
- [44] C-Linda User’s Guide & Reference Manual, Scientific Research Associates, 1992.
- [45] E. Shapiro, *Concurrent Prolog: Collected Papers*, The MIT Press, Cambridge, MA, 1987.
- [46] G. Sutcliffe, J. Pinakis, N. Lewins, Prolog-Linda: an embedding of Linda in muProlog. Tech. Report 89/14, Dept. of Computer Science, Univ. of Western Australia, 1989.